Introduction

This document contains the reports written by students in Swarthmore College Computer Science Senior Conference (CPSC-97), Autumn 2013. This course gives students the opportunity to delve deeply into a particular topic in computer science by reading a selection of research papers and completing an original research project.

This year's cohort completed 21 projects broadly related to the topic of dynamic software analysis. The 2013-2014 Swarthmore College Computer Science graduating class is:

Imoleayo Abel
Davis Ancona
Eliza Bailey
Peter Ballen
Lisa Bao
Ethan Bogdan
Alex Cannon
Jordan Cheney
Stella Cho
Jon Cronin
Tony Farias
Dane Fichter
Leah Foster
Madison Garcia
Josh Gluck
Samantha Goldstein
Adrien Guerard
Jackie Kay
Gabriel Khaselev
Kyle Knapp
Chris Lekas
Sophie Libkind
Zach Lockett-Streiff
Cynthia Ma
Chris Magnano
Brian Nadel
Taylor Nation
Kenny Ning
Melissa O'Connor
Danny Park
Sola Park
Yeayeun (Y.Y.) Park
Craig Pentrack
Mallory Pitser
Luis Ramirez
Kevin Roberts
Will Schneider
Mark Serrano
Aidan Shackleton
Aashish Srinivas
Danielle Sullivan
Kevin Terusaki
Hugh Troeger
Niels Verosky
Bowen (April) Wang
Jake Weiner
Elliot Weiser
Sam White
Yeab Wondimu
Yujie (Jack) Yang
Hongin Yun
Shimian (Sam) Zhang

The 2013-2014 Swarthmore College Computer Science Department

Lisa Meeden, Department Chair
Joshua Brody
Andrew Danner
Jeff Knerr
Tia Newhall
Bridget Rothera
Frances Ruiz
Ameet Soni
Jason Waterman
Kevin Webb
Richard Wicentowski
Benjamin Ylvisåker

Contents

TaintDroid Malware Categorizer

Imoleayo Abel Davis Ancona William Schneider
{iabel1, dancona1, wschnei1}@swarthmore.edu
Swarthmore College

Abstract

Mobile technology has become a huge part of society, and plays an ever increasing role in our daily lives. Our mobile devices house immense amounts of our personal data, including browser history, location, bank account information, and passwords. With such a large amount of sensitive information, it is surprising to know that the level of default security on these devices is significantly less than that of desktop computers. Many applications we use daily use our private data and, if implemented maliciously, could severely endanger our security. To prevent this from occurring and to add a layer of transparency to third party applications, we have implemented a TaintDroid-based Malware Categorizer. TaintDroid is an efficient, system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. To this software, we have appended a threat/incentive categorization component. With this system in place, users will receive a comprehensible output dictating the threat that has most likely infected their device. This output will supply a tangible, real life example to the market consumer, and avoid the acute technical details which do nothing to assist and educate the average user. This will empower users, making them more safety conscious and better informed of which applications can and can not be trusted.

1 Introduction

1.1 Motivation

As computing technology advances, more and more emphasis is being put on mobility. The ability to communicate, access data, and perform a variety of computing tasks while on the go has led mobility to become synonymous with efficiency. Unlike desktop computers which experienced slow and steady growth, the quick emergence of mobile technology has caused these devices to lag behind in terms of security. As *mobile* devices, they are used more frequently on unsafe public networks, they are more likely to be lost or stolen, and they house a majority of our personal information both public and private, making them inherently insecure. Because of this, mobile devices have become an enticing target for attackers and the malware they create.

The typical Android user does not take appropriate steps to ensure the safe use of applications on their device. Faced with a browser SSL warning message (which can indicate a phishing attack), over 70% of users clicked through and proceeded to a potentially dangerous website. However, when users were presented with a message that specifically warned against phishing or malware, clickthrough rates decreased to under 25% [1]. This indicates that users are more willing to respond to warnings that are specific and easily recognizable. Thus, for our project, we want the user to understand accurately how the applications they are using could potentially harm them without overwhelming them with technical and specific details which they are likely to ignore.

To help combat this influx of malware, found mostly in third-party applications, TaintDroid [4] [Figure 1] was developed to detect the data being sent off a mobile device by an application. This allows users to verify to some degree that the data being transmitted off of their mobile device is necessary for the functionality of that specific application, and not being used maliciously. However, this requires the user to have more than just common knowledge about cyber security, embedded systems, and computer science in general. To solve this problem and more efficiently use TaintDroid's output for the average user, we have created a TaintDroid-based malware categorizer. Our system takes the same input as TaintDroid (an application to review)

and outputs a possible threat type and or malicious, incentive-based attack that the application could be concealing.

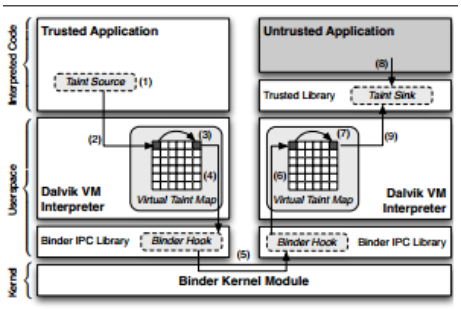

Figure 1: TaintDroid Architecture with Android

This added categorization better informs typical Android users how their data may be vulnerable, and which applications put them at higher risk. This may be a small step in terms of security, but increasing user awareness of malware is a fundamental step in its mitigation. The categorization could also be used for vetting processes in application stores. By cross referencing the possible threat types and attacks, applications could be given different security ratings and then be selectively admitted or denied when published for public download.

The overall motivation of this project is to close the gap between analysis and action in terms of security left open by TaintDroid. By paving the way for security-focused machine learning and categorization, the hope is to create a new category of user friendly tools to aid in the awareness and fight against mobile malware.

1.2 Background

The basis for our project is the application of a dynamic data tracing and analysis tool coupled with the functionality of a suite of classifying algorithms to categorize malware and effectively convey to the average user how their private information could be used against them.

The theoretical portion backing the classification algorithm of our system is comprised by research conducted on the current types of mobile malware plaguing today's application market. Because TaintDroid is only compatible with the Android system, our research was limited to Android malware. Our categorization identifies three overarching types of threats to group potentially hazardous applications [5]. These threat types can be broken down into smaller more specific instances, conveying generic examples of actual attacks based on the incentives of the malware creator. Thus, our theoretical classification basis can be represented by a hierarchy of threat based on type and incentive.

The framework of our classifier training set was developed based on this theoretical hierarchy. Each threat type matched to a broad set of TaintDroid output values, representing different types of private data, and each incentivised attack under the realm of a certain threat type was matched to a smaller subset of these values. This provided the foundation necessary to initialize the machine learning component. Specifically, the decision tree algorithm was chosen from the suite as our primary classification algorithm. Its efficiency with binary values created an ideal representation for our "targeted or not targeted" basis of valuing the sensitive data statuses.

Weka, the chosen machine learning tool described in **4.2** was integrated into the TaintDroid source code to provide a single, compiled tool in one executable program. The portion of the code used to display TaintDroid's output to the device screen was modified to pass the identity of the "tainted" values directly to our classifier. The classifier was modified to recycle its output back to TaintDroid, where the classified threat is conveyed to the user through a modified version of TaintDroid's display function. The user is now aware of the type of threat or specific kind of incentivised attack to which he or she is vulnerable.

1.3 Contribution

The field of security in mobile technology is not yet equatable to its static, desktop predecessors. Our project closes this gap by attempting to increase the level of familiarity with mobile security to that of desktop security. While TaintDroid on its own is a fantastic tool for malware analysis, it does not have the usability of a publicly available, consumer friendly program. By appending categorization functionality to an analytic foundation, our system creates the basis for a consumer friendly mobile security software similar to desktop security packages such as Norton Anti-Virus and McAfee. Through our addition of machine learning categorization, we develop a system which should be able to expand and evolve based on the threats it faces

like desktop anti-virus programs. Thus our malware categorization system promises to be a successful tool with which to educate and equip the average Android user to defend against post (and hopefully pre) market malware.

2 Malware, Grayware, Spyware

There are three main categories of potentially harmful Android software: grayware, personal spyware, and malware. Malware is categorized by being actively harmful to your device by hijacking your phone in order to send premium SMS messages or locking and ransoming your phone. Since it uses your device for nefarious purposes without your permission, malware is illegal in most jurisdictions. However, the legality of a piece of malicious software can depend upon the amount of damage done and the strictness of local laws.

Personal spyware is a category of application which runs silently on a device, collecting user activity data without being detected and then saving it to be read at a later time. It is designed to be undetectable, and so often does not transmit data over the network, making it just about impossible for TaintDroid to find. Spyware is legal so long as the devices user has consented to the application being run on their device, or in the case of a parent installing it on a childs phone. Using spyware to catch a cheating spouse or spy on someone is illegal and against the terms of use.

As the name indicates, grayware lies in a much more ambiguous legal area. Grayware may access your personal data and send it across the network, but it does not seek to harm the user or their device, and may spell out (however vaguely) its actions in the application's privacy policy. Although it may not be illegal, many Android users still prefer not to have their personal data sent to ad servers. Additionally, your data could be used for more nefarious purposes such as spying on your location or communication activity, either by a stalker or more recently a governmental agency. Your contact lists might also be sent out, which contain phone numbers and email addresses that can be sold to spammers.

This broad categorization of threats can then be further classified based off of the incentive the malicious software and previously known attacks. The most important information to the user is what the application is trying to accomplish so that the user can take steps to mitigate the effects, such as changing their bank passwords and uninstalling the application.

3 Idea

TaintDroid tracks sensitive information being sent out over the network by third-party applications from a host device and based on the data being sent, generates a notification. TaintDroid uses a TaintDroidNotify app to inform the user of the identity of the application sending data out of their phone. A sample notification is shown in figure 2 below. In addition to the identity of the application in question, the TaintDroidNotify app also provides a list of the taints (personal data) being sent out e.g IMEI, GPS Location, and so on. The notification also contains the destination IP address of the taint as well as the exact raw http request being sent from the phone. While all this information seems informative, we believe the http request is only useful to superusers - tech savvy smartphone owners. Our idea is to replace this displayed http request with a high level description of the potential harm to the device. This way, we avoid the risk of obscuring users who might be led to doubt the validity of TaintDroid if they don't understand what the information in the http request means. Also, having a high-level description like 'App x is most likely going to steal your bank information' creates a sense of importance that is immediately understandeable by any user irrespective of technical ability.

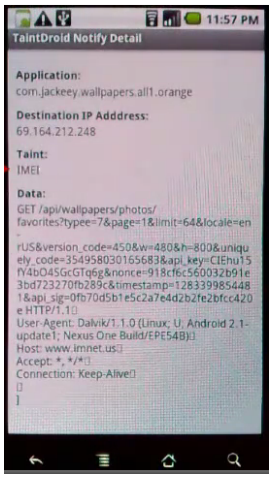

Figure 2: Snapshot of sample TaintDroid notification message

Our goal is to classify the potential harm posed to a device by a third party application. The classification will be done using the Java-based machine learning suite−Weka.

4 Classification

4.1 Classification Model

To come up with a classification model, information from published papers [5], [6], [2], and [3] about the behavior of mobile malware was used to create a training set. Specifically, information was gathered about what data is required to pose certain harm to mobile devices. The pieces of data used for this model were the 16 variables tracked by TaintDroid listed below:

Device Serial Number	Location
Address Book (ContactsProvider)	SMS
Microphone Input	Phone Number
NET-Based Location	GPS Location
Last known Location	IMEI
Browser History	IMSI
ICCID (SIM Card Identifier)	Camera
User account information	Accelerometer

From findings about the behavior of malware, a classification model was designed. As an example, the training set for classification will look somewhat like that shown below where for every kind of mobile threat, there is a corresponding combinations of tainted data that could be used to effect such a threat. There will be more categories and multiple instances of data combinations for the same threat in the actual training set. The hierarchy can be seen here:

Spyware

- Selling user Information: IMEI, IMSI, Device Serial Number
- Spouse/Parent Tracking: Location*, Microphone, Accelerometer, SMS, Browser History, Camera.

Malware

- Premium Rate Call: SMS, Phone Number
- SMS Spam: SMS, Phone Number, Contact List
- Ransom: IMEI, Browser History
- Stealing User Credentials: User Account Information
- Selling User Information: Browser History, Contact Information, Location*, User Account Information, IMSI, ICCID, IMEI

Grayware/Adware

- Targeted Ad: Location, IMEI, IMSI, ICCID, User Account Information, Browser History, Device Serial Number, Phone Number.

* Location is any of GPS, NET-based or Last Known locations.

4.2 Weka

Weka is an open source machine learning and data mining suite written in Java and developed at the University of Waikato, New Zealand. It contains a collection of visualization tools and algorithms for data analysis and predictive modeling, together with graphical user interfaces. It also allows integration with Java projects in cases where this functionality is desired. Weka classifiers take as input a file that contains the list of attributes and the possible values for each attribute. Weka input files also contain a set of examples with which the categorization model is trained. An example is a list of attribute values with the target attribute value being last. The image below shows an example input file for Weka classification:

```
@relation potentialHarm

@attribute Location {0, 1}
@attribute Last_known_Location {0, 1}
@attribute camera {0, 1}
@attribute accelerometer {0, 1}
@attribute SMS {0, 1}
@attribute PhoneNumber {0, 1}
@attribute IMEI {0, 1}
@attribute IMSI {0, 1}
@attribute Device_serial_number {0, 1}
@attribute browser_history {0, 1}
@attribute harm {premiumCall, ransom, grayware, SMSspam}

@data
0,0,0,0,1,1,0,0,0,0,premiumCall
0,0,0,0,0,0,1,0,0,1,ransom
...
...
```

Figure 3: Sample input file for Weka Categorization

4.3 Decision Trees

Weka implements a large number of classification algorithms, but the choice for this project was the decision tree algorithm. Decision trees are directed

tree-like graphs of decisions and their possible consequences. Nodes in a decision tree represent attributes and edges emerging from a node correspond to the possible values of the attribute represented by that node. Leaf nodes are target attributes or the classes/labels. In building a decision tree, the root node is chosen to be the attribute that splits the training set in such a way that the entropy of the training set is minimum. Vaguely speaking, the root node is chosen to be the attribute that divides the dataset as evenly as possible such that the training set for each subtree from the root node is as small as possible. When a new example is to be classified by a decision tree, the tree is traversed starting from the root node where the value—in the example to be classified—of the attribute represented by the root node is compared to the branches that emerge from the root node. Based on this value, the traversal of the tree is continued at the corresponding subtree. This process is continued until a leaf node (which is the appropriate classification for the example) is reached. An example decision tree to clasify/decide if a day is suitable for playing tennis is shown.

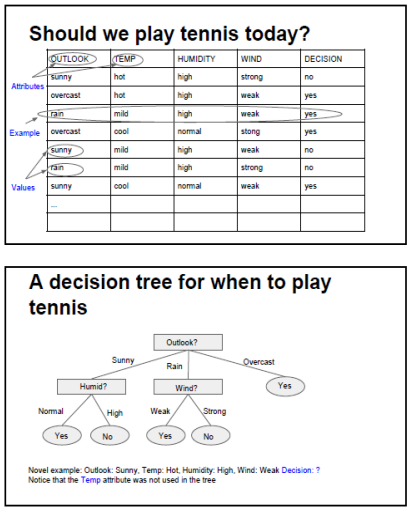

Figure 4: Decision tree for deciding whether the weather is suitable for playing tennis. *Excerpt from Lisa Meeden's slides on Decision Trees in Artificial Intelligence course at Swarthmore College; Fall 2013.*

For our classification problem, the value of the attributes—the sixteen pieces of data tracked by TaintDroid—would be binary. The value for an attribute in an application's example would be 1 if the application sent out data for that attribute and 0 otherwise. Since the attributes have binary values, the choice of a decision tree was natural as decision trees are suited for binary-valued attributes. Decision trees also have the advantage of being easy to understand and being less computationally intensive as compared with other popular categorization/machine learning models, such as artificial neural networks.

5 Evaluation

5.1 Test Malware

A few malicious Android applications were handpicked from an online malware repository. These applications were designed specifically to export data from a host device to an external server over the network. A screenshot of one such application *gi60s*—'gone in 60 seconds' running is shown in Figure 5 below. *gi60s* sends user's contacts, messages, recent calls and browser history to an external server in under 60 seconds. To ensure there is data to be sent, dummy contacts were created in the contact list of the emulator, browser history was generated on the emulator, and dummy texts were sent to the emulator via telnet on port <5554>.

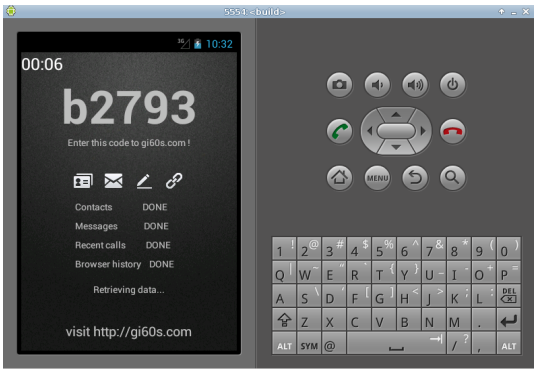

Figure 5: Snapshot of test application *gonein60seconds* running on the Android emulator.

5.2 Results

The performance of a classification model is only as good as the quality of the information in the training set used in creating the classifier. The limit of the performance of our classification system depends on the accuracy of the model we create. While it is impossible to perfectly predict what an application is going to do with data sent out of a mobile device, the intention of this project was to predict as accurately as possible—based on past empirical findings—what harm a mobile device could fall victim to. However, the primary goal of this project was to

have the generated threat categorization create a sense of imminent danger that would cause users to take notifications of malware threats more seriously.

We were able to successfully acquire and build the TaintDroid source code without errors. The image below shows TaintDroid running on the Android emulator.

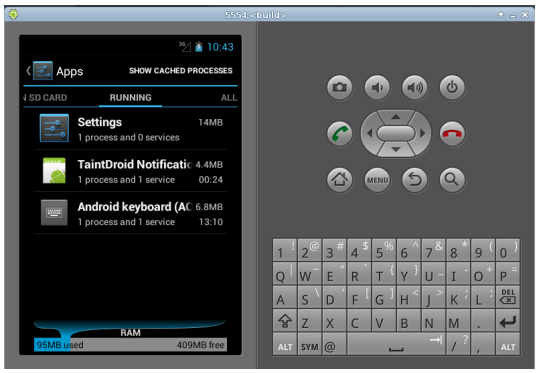

Figure 6: TaintDroid running as a background process on emulator.

At a high level, the functionality of TaintDroid is divided into two steps. First, there is a TaintDroid-NotifyService [NotifyService] that does the actual taint tracking and secondly there is a TaintDroid-NotifyController that handles the GUI for the notification. NotifyService contains two subclasses: a Producer and a Consumer class. The Producer class is responsible for populating a logfile from which the Consumer class reads and sends data to the TanitDroidController for the GUI display. We noticed during development that the Consumer class had bugs that hindered the reading of data from the logfile which consequently prevented TaintDroid from generating notifications. Upon further investigation from TaintDroid developer forums, it was gathered that there were some incompatibilties with the Android 4.1 file I/O. On checking the Android *logcat*, we confirmed the problem with the Consumer class which is highlighted in Figure 8 below. Figure 7 shows that the Controller was successfully instantiated when the TaintDroid app is started:

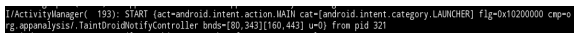

Figure 7: Android *logcat* of background processes on the emulator.

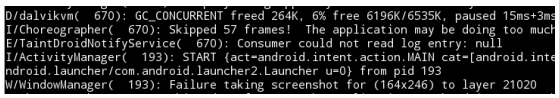

Figure 8: Android *logcat* of background processes on the emulator.

The integration of Weka to TaintDroid was successful as we were able to rebuild the TaintDroid source without errors after including Weka libraries and adding Weka to the build path of the TaintDroid project. The behavior of TaintDroid was consistent as before the integration with Weka. We also tested our Weka classifier with a simple Java program to make sure we can use Weka libraries from Java code without a GUI. Weka is a GUI-based suite and full functionality from within Java source code isn't guaranteed.

The fact that NotifyService is triggered on launching a malicious application and not with regular applications (that do not use the network) confirms that the integration of TaintDroid with Weka did not hinder the functionality of TaintDroid. In addition, a replica implementation of the Weka classifier in a simple Java program worked without errors confirming that the concept of our project was successful. As mentioned earlier, the accuracy of the Weka classification is only as good as the malware behavior model that was generated from our findings in published research work, but our main objective of user friendly warning is functionals.

6 Conclusion

Our system, although not fully functional, was able to detect and categorize threats on our emulated Android device. With this capability, an Android malware attack can be quickly detected and assessed based on its threat level. This sort of power could prove tremendously helpful to average users as they seek to keep their devices safe. More seriously, damaging apps could be isolated and removed from infected phones. Since our program uses machine learning techniques, it can learn and adapt to different attacks it faces, helping to keep it current with new generations of malware which are constantly adapting to stay ahead of Android security software. With enough training and keeping current with the malware attacks that spring up, software such as ours should be able to be a potent tool in the battle between mobile malware and their victims.

7 Future Work

Given more time and resources, the next step in our project would be to implement our malware categorizer on an Android phone. Because of numerous issues with integrating the WEKA software

into a mobile environment, we were forced to run our code on an Android emulator. Changes to the functionality of our program could include taking into account the level of permissions willingly granted to applications that we test. This would help avoid false positives, since we do not want the Google Maps application to trigger a warning for sending out the phones location.

If we were to scale up this app for use by the general public, a number of modifications to the TaintDroid source code would also be required. Currently, TaintDroid only runs on Android versions up to 4.1, while the current Android release is 4.4. Also, we would need to broaden the scope of devices that TaintDroid runs on, which is currently limited to just two phones: the Nexus S and the Galaxy Nexus. Additionally, installing TaintDroid requires the user to root the device, which bypasses many of the default Android security features and can make phones even more vulnerable to attacks. We would want our software to be downloadable and installable straight from the Android market to maximize its user-friendliness. Once our software is up and running and is known and trusted by Android users and developers, it can be used on existing applicationss to impart a security rating based on how dangerous it is categorized by our software. It is our hope that such a system would encourage Android users to think more carefully about how their data is used and hold developers to a higher standard for security and privacy.

8 Acknowledgments

Our paper draws heavily from the original TaintDroid paper published by William Enck et al. We got in touch with the original authors and maintainers of TaintDroid to aid us in converting the program for our use and are indebted to them for their help with using the TaintDroid API. We also drew on the paper All Your Droid Are Belong To Us: A Survey of Current Android attacks for inspiration and information on the current state of mobile, especially Android malware. A Survey of Mobile Malware in the Wild was also a good source of information, and served as the main basis for our classifier in dividing up different malware attacks. Another paper we cited was Behavioral Detection of Malware on Mobile Handsets, which provided us with a lot of the information on how to go about using a machine learning classifier to detect and categorize malware.

References

[1] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *USENIX Security Symposium*, 2013.

[2] Michael Becher, Felix C. Freiling, Johannes Hoffmann, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 96–111, Washington, DC, USA, 2011. IEEE Computer Society.

[3] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 225–238, New York, NY, USA, 2008. ACM.

[4] William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdoid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Stmposium on Operating Systems Design and Implementation*, OSDI '10, 2010.

[5] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[6] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

Performance Evaluation of Concurrency Primitives in Charcoal

Jacqueline Kay (jkay1) Joshua Gluck (jgluck2) Lisa Bao (lbao1)

20 December 2013

Abstract

Most modern software has some asynchronous interaction with its environment. We present an alternative solution to the two existing major paradigms, event handlers and parallel threads, for managing asynchronous events. Event-driven programming in the loop/handler pattern can simulate concurrency, but it scales poorly as task complexity increases. Multithreading allows the programmer to specify true simultaneous control flows, but parallel programming is notoriously bug-prone.

The Charcoal programming language addresses these issues through a pseudo-preemptive threading framework called activities. We evaluate the comparative performance of different implementations of the yield concurrency primitive for activities in Charcoal and find that its speed will permit the Charcoal compiler to insert implicit yield statements without unduly increasing program overhead.

1 Introduction

Mainstream software today is expected to successfully handle asynchronous events. For instance, a software system often needs to wait for user input or receive network packets. To date, solutions for handling asynchronicity have fallen into two general categories: event handlers/call-back functions and concurrent, parallelizable threads.

One way to wait for asynchronous events without continuous blocking is to use an event loop-and-handler pattern, which checks for newly arrived events and calls the appropriate handler function. However, when multiple anticipated events can coincide with each other—such as keyboard input, mouse clicks, and network signals—the event handler design pattern becomes increasingly complex and difficult to understand. Moreover, the control flow of actually responding to an event must be handled in a callback function, which may be forced to duplicate code implemented elsewhere in the program control flow.

Alternatively, multiple threads or processes can be created to wait for events while efficiently sharing processor time so that other parts of the program can still run. This type of concurrent multithreading has both the advantages and the disadvantages of parallel programming. Although threads form an intuitive control flow model, in practice they spawn subtle, hard-to-detect bugs such as race conditions.

Most threading models currently in use are forms of concurrent threads which run in parallel, but an alternative model also exists: cooperative threading. Cooperative threads maintain control until they explicitly yield to another thread, and shared data structures can only be accessed once control is yielded. However, in a traditional cooperative threading framework, the programmer must manually specify context switches between threads. Charcoal is a new C-dialect programming language under active development which seeks to address this problem by implementing a hybrid, pseudo-preemptive threading framework.

In this paper, we present an evaluation of the performance of the yield concurrency primitive in Charcoal. Unlike most cooperative threads, Charcoal's compiler inserts implicit yield statements after code blocks; while the programmer has some control over yielding, the purpose of this compiler behavior is to automate context switching logic in cooperative threads. Therefore, a typical Charcoal program expects to frequently invoke yield, making the efficiency of the yield call very important.

We implement multiple versions of yield and compare their speed and efficiency in order to confirm that the Charcoal compiler will be able to insert implicit yield statements without greatly impacting program overhead. Moreover, we determine the optimal frequency of calling yield while minimizing overhead. The results of our evaluation will be influential to the design and evolution of the Charcoal language.

The next section of this paper summarizes background context and related work. Section 3 introduces Charcoal and elaborates on its driving motivations. We evaluate our implementation of yield and present the results of our tests in Section 4. Finally, we discuss final conclusions and future work in Section 5.

2 Background

In event-driven programming, an event loop periodically checks for input messages or events. For instance, an event such as a key press may trigger a callback function which responds to this input by writing a particular corresponding character to the screen. This technique is useful for reacting simply and appropriately to multiple sources of input which may occur asynchronously. However, it also presents a major problem: all computation must be completed in callback functions, while the event loop is reserved for waiting on the next event. This control flow can lead to messy, unconventional, and inefficient code as the programmer attempts to work around the enforced structure [9].

Event-driven programs often give the illusion of concurrency, but in fact they are not multithreaded [5]. Attempting to parallelize an event-driven program, such as by running event handlers on multiple processors, would significantly warp the intended control flow of the program. Multithreading addresses some of the problems of event-driven programming by allowing the programmer to express multiple, simultaneous flows, rather than simulating concurrency with a series of callback functions. However, parallel programs are often more difficult to write because the programmer must handle race conditions and concurrent accesses leading to data races; even so-called "benign" data races can result in program failures [3, 7].

Both of these solutions solve the problem of asynchronous event handling while also ensuring that processors do not spend a significant portion of their time waiting idle. However,

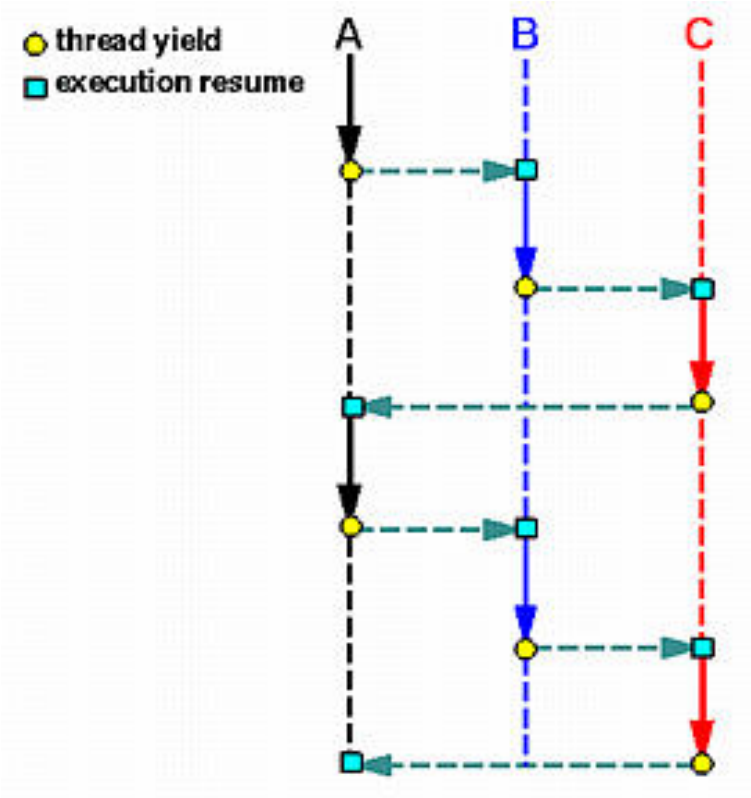

Figure 1: A diagram of the control flow of 3 cooperative threads yielding to each other [8].

event loop/handler patterns can lead to complicated control flow, since long-running tasks must be broken up into multiple event handling invocations. While the design pattern of multiple threads or processes works well on a single CPU, the ability of threads to run in parallel—combined with the advent of multiprocessor machines—has led to significantly more bug-prone thread-based event handling due to the inherently complex nature of parallel processing [6]. Prior research to solve this problem has focused on making parallel programs more reliable, such as through deterministic multithreading [2] or stable multithreading systems [4, 10].

In contrast to the conventional, concurrent threading framework, cooperative threads were designed with shared data access in mind. While conventional threads guarantee that each thread will eventually yield control to another thread, cooperative threads guarantee that a thread will never unexpectedly yield control to another thread [1]. Data structures shared between cooperative threads can only be accessed by other threads when the current thread gives up control of these resources. Only one cooperative thread runs at a time, and the programmer must handle context switching between threads by placing explicit yield statements throughout his or her code.

Cooperative threads are equivalent to event-driven programs in their safety from data races. In fact, a program with cooperative threads can be converted into an equivalent event-driven program by replacing every block call with a return to the event loop, although this process requires first-class continuations (an exactly specified order of instructions). Thus, cooperative threads are an expressive alternative to event-driven programming. However,

a cooperative multithreading approach depends on the programmer spending the time to determine when threads should yield control. Although cooperative threads have existed for a relatively long time, they have never seen the same popularity as event handlers or preemptive threads. We are not aware of any related research which attempts to combine the advantages of cooperative threading with the automatic resource-sharing of parallel threading, as the Charcoal thread framework is designed to do.

3 Charcoal

Our work implements part of the Charcoal programming language, a C dialect under active development which employs *activities* rather than threads [11]. Activities represent a middle ground in the choice between preemptive and cooperative thread programming. They are implemented in a manner similar to cooperative threads but are equally influenced by the functionality of preemptive threads, since the compiler inserts a number of implicit yield invocations throughout the program. In other words, Charcoal provides cooperative threads without requiring the programmer to explicitly place yield statements in his or her code.

We focus on implementing, profiling, and optimizing different implementations of the Charcoal yield function in order to make it as efficient as possible. Three different variables affect whether or not a Charcoal activity will context-switch (i.e., give up control of processing power) to another activity when yield is called.

1. A programmer-controlled *unyielding* flag can prevent context switches from occurring. It allows the programmer to override the compiler by allowing one activity to run uninterrupted for long periods of time. Unyielding is an important synchronicity primitive because it permits large atomic sections of code.

2. After the passage of a certain length of time, a context switch should occur. Thus, yield needs to be invoked at just the right frequency; too often will lead to large overhead, whereas too rarely will result in the resource starvation of other threads.

3. Certain kinds of interruptions, such as a signal delivery, should trigger a context-switching yield even if the activity has only been running for a short period of time.

Our first version of yield uses a pair of activity-shared atomic integers, as well as an alarm timer, to achieve this functionality. The first atomic integer, normally set to 0, is incremented when the unyielding flag is set and decremented after the unyielding block is finished. The second atomic integer is initialized to 0 and set to 1 by the alarm after a compiler-specified period of time. For a yield to context-switch, the unyielding flag must not have been set and the timeout integer must have been incremented.

Our second version of yield improves on the first by using only a single activity with a shared atomic integer and alarm timer. This atomic integer is initialized to 1, incremented by the unyielding flag (and subsequently decremented after an unyielding block of code), and decremented by the alarm after the same pre-specified period of time. This implementation combines the functionality of both atomic integers in the first implementation, ostensibly lowering the computation costs of a yield statement.

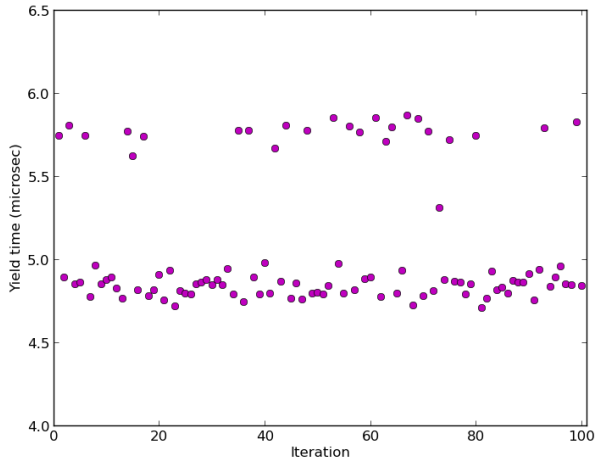

Figure 2: Results over 100 iterations for context-switching yield.

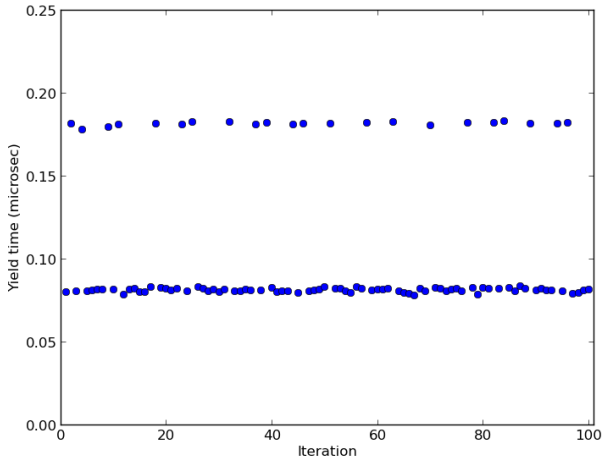

Figure 3: Results over 100 iterations for non-context-switching yield.

4 Evaluation

In this section, we give an evaluation of our yield implementation in Charcoal. Our experimental study has two main focuses. First, we contrast two yield implementations for correctness and efficiency. The importance of correctness, both forcing only a single activity to run and allowing yields between activities, is obvious. Because both context-switching and non-context-switching yields will incur additional overhead, a more efficient yield function will decrease the runtime cost of using Charcoal.

Secondly, we measure the frequency of yield placement in code with varying overhead costs over the program runtime. These measurements provide useful information for balancing potential thread starvation from infrequently calling yield against the increased overall runtime of a program implemented in Charcoal. By illustrating a range of frequency-overhead relationships, we allow software developers to choose the relationship between yield frequency and overhead that best fits their use case.

4.1 Yield Runtime Characteristics

Both implementations of yield were successful in mediating processor access among multiple activities. By running the same program with and without 200,000 context-switching yield calls, we found that the context-switching version of our first yield function took an average of 5-6 microseconds to run. In comparison, the non-context-switching yield took approximately 50-60 nanoseconds, which confirms our suspicion that a successful context switch takes up the majority of the additional runtime incurred by yield. The results of these tests are shown in Figures 2 and 3.

Our second yield implementation was significantly more efficient. The context-switching version of yield took an average of 5 microseconds to run, as seen in Figure 4. Similarly, Figure 5 shows the results for the non-context-switching version of yield, which took 30-40 nanoseconds—an approximately 50% decrease in runtime compared to the first version.

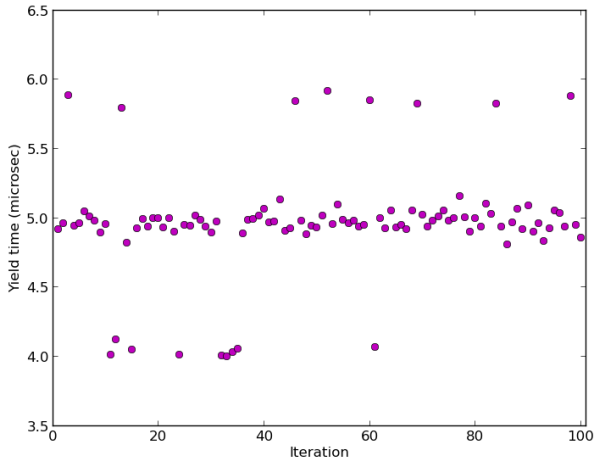

Figure 4: Results over 100 iterations for context-switching efficient yield.

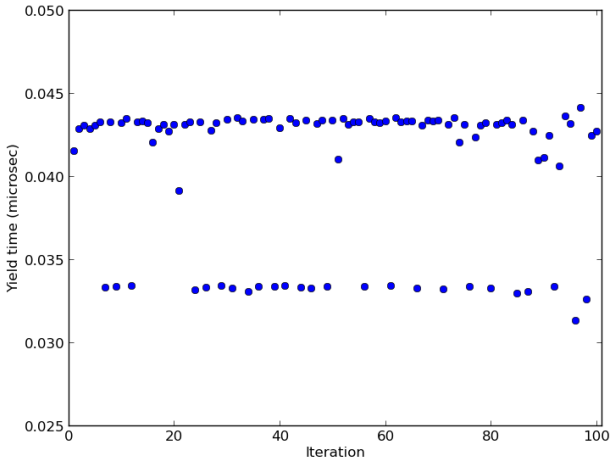

Figure 5: Results over 100 iterations for non-context-switching efficient yield.

Given these preliminary results, we chose to perform our remaining tests on the second, more efficient yield implementation.

4.2 Yield Placement

In order to characterize the overhead of repeatedly calling yield in a Charcoal program, we constructed a nested loop test as shown in Figure 6. This test simulates an unyielding inner loop of m iterations within a larger loop, for a total of nm loop iterations. The function strcpy was chosen because it is commonly used in C programming and incurs a relatively large overhead.

```
for (i = 0; i < n; i++){
    for (j = 0; j < m; j++){
        strcpy(a, b, buffer_size);
    }
    yield();
}
```

Figure 6: The nested loop test for yield overhead.

We ran the test with 1 million total calls to strcpy and varied the number of iterations of the inner loop. We defined the overhead of yield as

$$\frac{total\_time - noyield\_time}{total\_time}$$

where *total_time* is the elapsed time of the nested loop test and *noyield_time* is the elapsed time of nm calls to strcpy (with no yields).

As Figure 7 indicates, the overhead of yield is inordinately high if it is called after every strcpy. The overhead declines exponentially with the number of inner loop iterations. Qualitatively, this result confirms our intuition that there exists a dropoff point where reducing the overhead of yield by calling it less often does not benefit the program's performance and moreover introduces the issue of thread starvation. Quantitatively, we found the dropoff point to be at $n = 1000$.

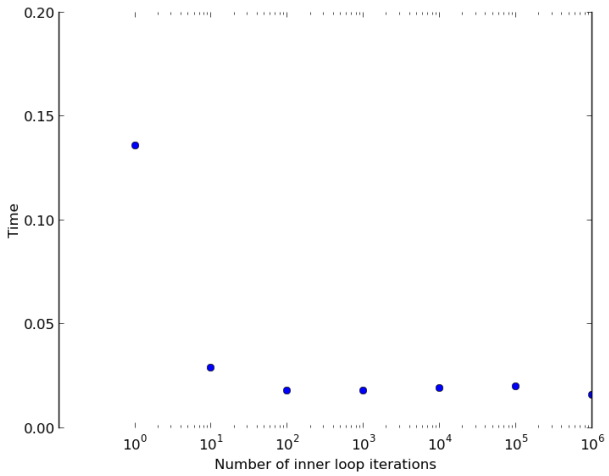

Figure 7: Yield overhead as the number of loop iterations increases for the memcpy test.

5 Conclusions and Future Work

From these experiments, we conclude that the placement of implicit yield statements by the compiler entails a larger number of yield statements than if the programmer writes explicit yield statements, due to the necessity of ensuring that thread starvation does not occur without taking domain knowledge into account. The relatively low cost of context-switching yields—and even more importantly, the nanosecond-range cost of non-context-switching yields—demonstrates that the cost of individual yields can be amortized to a minimal overhead when called frequently. Together, these results suggest the viability of a cooperative threading framework based on compiler-introduced implicit yield statements. Such a framework will allow programmers to achieve the benefits of a cooperative threading paradigm without the burden of determining the best places for yield statements, which is a daunting task for complex programs with many threads.

The preliminary success of the yield concurrency primitive points toward a number of avenues for future research. First, we would like to continue testing the optimal frequency of yield calls by using a multithreaded production-level code base to simulate yield patterns. Such a rigorous test would confirm our current results based on the strcpy and memcpy functions. Furthermore, other concurrency primitives such as mutex also need to be thoughtfully designed and carefully tested.

There remains much to be done on Charcoal before it can launch as a programming language for intelligent cooperative threading. We have outlined here our contribution, confirming the viability of the yield implementation, and we hope that our work will inspire others to pursue a similar plan of research with respect to pseudo-preemptive/semi-cooperative threading frameworks.

References

[1] Martín Abadi and Gordon Plotkin. A model of cooperative threads. *POPL*, pages 29–40, 2009.

[2] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. *ASPLOS*, pages 53–64, 2010.

[3] Hans-J. Boehm. How to miscompile programs with benign data races. *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism*, 2011.

[4] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. *SOSP*, pages 388–405, 2013.

[5] Andreas Gustafsson. Threads without the pain. *Queue*, 3(9):34–41, 2005.

[6] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *ASPLOS*, pages 329–39, 2008.

[7] John Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Winter Technical Conference*, January 1996.

[8] C. Shene. Basic thread management. `http://www.cs.mtu.edu/~shene/NSF-3/e-Book/FUNDAMENTALS/thread-management.html`, 2013.

[9] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). *Proceedings of HotOS IX*, 2003.

[10] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. Determinism is not enough: Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 2014.

[11] Benjamin Ylvisaker. Charcoal: Easier concurrency for application developers. `http://charcoal-lang.org/`, 2013.

Auto-tuning Sorting Algorithms

Sam White, Yeab Wondimu, Kyle Knapp

December 20, 2013

Abstract

We present an auto-tuning system for sorting algorithms that makes use of the various levels of parallelism present in modern systems. Auto-tuning involves empirically searching through a parameter space to choose a best or very good combination of parameters. Our empirical search algorithm considers the input data's size and standard deviation, as well as different hardware characteristics and multiple sorting algorithms. Hardware parameters involved are CPU cores and caches, along with the presence of GPU accelerators and other nodes connected via the network. The sorting algorithms that we implement are quick sort, merge sort, and bitonic sort.

The novelty of our sorting library lies in adaptivity to different degrees of hardware parallelism. If the parameters related to parallelism are not optimally chosen, there can be unnecessary slowdown resulting from underutilization or overutilization of resources. Our results suggest that our auto-tuning framework adapts to the complexities inherent in modern systems. For example, our results show running the best parameter combination from a sixteen core machine on a four core machine results in a run-time about 20% slower than the best parameters chosen by our library for that machine. Further, if we extrapolate our results to data sets that are significantly larger, it is reasonable to expect that the performance degradation expands to minutes or even hours.

1 Introduction

In Section 1.1, we will discuss background information related to previous uses of auto-tuning, how we auto-tune sorting algorithms, and the importance of sorting large data and thus the importance of using auto-tuning to sort as fast as possible. In section 2.2, we will discuss previous work related to auto-tuning sorting libraries.

1.1 Background

Sorting is one of the most performance-critical and well-researched families of algorithms in computer science. Automatic performance tuning, or auto-tuning, is a more emergent topic, but one that is now rising in popularity. Research into both general auto-tuning and domain-specific auto-tuning has grown in the last decade as a way to keep up with the complexity of modern hardware systems. Projects like ATLAS, FFTW, and SPIRAL tune numerical algorithms for different architectures and memory hierarchies for high performance applications. The Active Harmony project is an example of a generalized auto-tuning framework. Our project applies the methods of auto-tuning to sorting algorithms in order to gain high performance across various system architectures.

Auto-tuning involves two basic steps: generation of empirical data using various combinations of parameters, and searching through this parameter space in order to find a near optimal combination. The parameters we examine are the randomness, or standard deviation, of the input data as well as its size. We also take into consideration the cache size, number of cores available, and whether we are sorting using the CPU, GPU, or the network. Our library provides the user with a single API to a set of sorting algorithms that are performance tested and tuned based on the input data and the underlying machine architecture. Our system requires install-time tuning but further builds up its database with each call to it.

Our project is motivated by the idea that auto-tuning sorting algorithms can achieve considerable speedup over naive implementations by estimating

the optimal parameters for the sorting of a given data set. Especially when sorting large sets of data, knowing the optimal parameters greatly decreases the run-time and saves the user the time it would take to figure out what the optimal parameters are by themselves and port their solution to different systems. The overhead of auto-tuning can be kept low enough that picking the right algorithms is worth the extra cost of finding it.

Sorting large sets of data is integral to several real world applications. Google's MapReduce (or Apache's Hadoop), a distributed programming model for processing large data sets, uses a map function that processes a single key-value pair and generates many intermediate pairs. The map step does this for all key-value pairs and then the reduce function combines all values associated with the same intermediate key. MapReduce may be used to produce various representations of the graph structure of web documents, generate summaries of the number of pages crawled per web host, or determine the set of the most frequent queries a search engine gets per day. Each of these applications requires MapReduce to merge large inputs of key-value pairs which is done by sorting them by key in its intermediate reduce step [2]. Another application that does a similar sorting process is used by the Environmental Protection Agency (EPA). The Environmental Monitoring and Assessment Program, also known as EMAP, monitors the availability and status of a given natural resource over a large areas. EMAP works by doing systematic sampling of smaller 1.8km areas and representing these samples as key-value pairs. When all of the key-value pairs that represent these samples are sorted, it gives the EPA some data about the status of a resource over large areas [1]. Some other examples include data mining or web indexing, which involves indexing the content of websites for use by search engines. [3] All of these examples show the importance of sorting large sets of data efficiently.

1.2 Related Work

Research into auto-tuning domain-specific problems has been less extensive the further one gets from numerical methods and linear algebra libraries. Xiamong Li, Maria Jesus Garzaran, and David Padua have authored two papers on auto-tuned sorting libraries.

Their first paper, "A Dynamically Tuned Sorting Library" [4], focuses on examining the inputs to sorting algorithms and using an empirical search algorithm to figure out what combinations of parameters generate optimal performance. They focus on quick sort, multi-way merge sort and a cache-conscious radix sort.

More specifically, Li et al. showed that both architectural factors and the input data's characteristics were important factors in the run-time of sorting algorithms. Architectural factors they studied included cache size, cache line size, and the number of registers available on-chip. Their library preliminarily scanned the data being sorted to determine its size and entropy, a measure related to standard deviation. Their results showed that some sorting algorithms perform better on data with less of a distribution, others do better with a wider distribution, and still others are less sensitive to the input data's randomness.

Their second paper, "Optimizing Sorting with Genetic Algorithms"[5], concentrates more on the Artificial Intelligence concepts used to speed up the empirical search phase of auto-tuning. They use a genetic algorithm to search through the many combinations of parameters that could be used in order to adaptively partition the data and select the best sorting routines for each partition. In this way, they can use a different sorting algorithm for each partition, based on that partition's characteristics.

For each sorting algorithm, the authors use a library generator that searches through algorithm-specific parameters to find the optimal parameters for that algorithm. Some of the examples of parameters that the library generator searches are the value of the pivot used in the first phase of quick sort and the size and number of children in the heap used by multi-way merge sort.

Li, Garzaran, and Padua did not implement any parallel sorting algorithms, however, so our project in some ways extends their work by considering the optimal combinations of different levels of parallelism. By including GPU and over-the-the network solutions in our library, we get a much higher level of potential parallelism in our system. Additionally, the offloading point for moving data over the network or to a GPU from a single core is difficult to pinpoint in modern systems, making them good candidates for auto-tuning. Auto-tuning supplies the adaptability required to deal with all of these

hardware complexities.

2 Our Idea

2.1 Tuning Parameters

Our system auto-tunes three parameters for sorting the data: the type of sorting algorithm, the number of threads used to sort the data, and whether the data will be sorted over the network. Our library implements three sorting algorithms: quick sort, merge sort, and bitonic sort. Quick sort and merge sort are performed on the CPU, while bitonic sort is only performed on the machine's GPU. Bitonic sort is an in-place, comparison-based implementation of a sorting network. Bitonic sort has a runtime of $O(nlog^2(n))$. Therefore, it is not optimal for running on a CPU when compared to quick sort and merge sort, which both run in $O(nlog(n))$ time. However, bitonic sort is an embarrassingly parallel algorithm, so it runs faster on many-core architectures that are highly parallel, such as GPUs. [6] We decided to choose quick sort and merge sort because previous research had shown the two sorting algorithms differ in performance as the size and the standard deviation of the input data changes[4] and are easily parallelizable. Other algorithms such as radix sort can be included in our library in the future with ease, as our library is designed to be modular.

Our system allows for the use of 2^n threads to sort the data on multicore processors. These threads will be CPU threads if the algorithm being run is quick sort or merge sort. If the algorithm used is bitonic sort, then GPU threads are used. The CPU threads are implemented using the pthreads library and our bitonic sort is implemented using CUDA.

As to the use of the network to sort the data, this parameter is binary, either the data is sorted over the network or it is not sorted over the network. If the network is used, we scatter the input data over the network, sort each chunk of data on the individual nodes (using CPU threads or the GPU), and then merge the chunks back together in a tournament fashion. Our network merge is implemented using the MPI library.

2.2 Training

In order to pick the appropriate parameter combinations, the system must have some knowledge of what parameters work best for that machine. To generate such knowledge the system creates a database of text files that record how long it took a specific machine to sort some set of data using one of the parameter combinations discussed in Section 2.1. The information in each of the text files can be classified into three groups: machine specifications, data characteristics, and sorting parameters. The machine specifications include the number of cores, the cache size, and the GPU size. The data characteristics include the size of the data and the standard deviation of the data. The sorting parameters include the type of algorithm used, the number of the threads used, and whether it was sorted over the network.

To generate a large database of these tuning files, our library runs every possible parameter combination on many sets of data that vary in both size and standard deviation during an installation period (this took about an hour to an hour and fifteen minutes for all systems we used). We repeated this process on all the different machines available to us in order to obtain some variety in the number of cores, the size of the cache, and the size of the GPU. These text files are saved in the format of Figure 1. The content of the data files is exactly the same as its file name. The only difference is the parameters are listed line by line, as opposed to being separated by underscores.

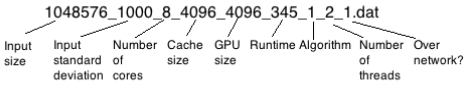

Figure 1: Example data file

2.3 Parameter Search

With the database generated, the system is able to search through it to determine the best sorting parameters to use given the machine's characteristics and the data's characteristics. Our system's parameter search is depicted in Figure 2. Because

the parameter space is not too large, we use a one-at-a-time filtering search procedure.

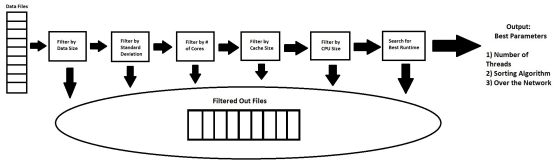

Figure 2: Depiction of parameter search

More specifically, the system first determines the size and the standard deviation of the data by sampling over the list. It then scans the text files in the database, searching for the text files whose data size is closest to that of the data to be sorted. The text files that do not have this closest data size are immediately removed from the list of text files to continue searching through. After filtering by the size of the data, the system searches through the remaining text files for the text files with closest standard deviation to that of the data to be sorted. It then removes the files that do not have this closest standard deviation as well. This search is continued by filtering through text files that best match the host machine's specifications. The system filters first by number of cores, then by the size of the cache, and then by the size of the GPU. Once the text files have been filtered based on machine specifications, the system searches through the remaining files in order to find the shortest run-time. The system then uses the parameters of the text file with this shortest run-time to sort the user's input data.

3 Evaluation

In this section, we present the results from running our system on machines with different specifications. In Section 3.1, we discuss the enviromental setup of our system on various machines. In Section 3.2, we present data pertaining to the performance of our system on machines that differ in specifications. In Section 3.3, we make a baseline comparison between our system and C's stdlib qsort().

3.1 Environmental Setup

The two types of machines we ran our tests on are listed in Table 1. Note that when we ran our system over the network, all machines being used belonged to the same group in terms of machine type. Both of these types of machines ran a version of the Linux operating system.

Unless the data files, as discussed in Section 2.1, had already been imported from previous trainings, the data files had to be generated by training it over our set of input files. The input files varied in both size and standard deviation. The size of the input files ranged from 16 KB to 512 MB. The standard deviation ranged from 5 to 100,000. Overall for a given training session, we used 35 input files, each with a different size and standard deviation. Our system took an hour and fifteen minutes to fully train our system on machines of the classification Machine Type 1 from Table 1. For machines of the classification Machine Type 2, it took only an hour to fully train our system for that type of machine.

3.2 Performance of System

After training our system on both types of machines, we used our system to sort data of varying size and standard deviation on both types of machines. We first tested our system on the machine classified as Machine Type 1 from Table 1. We present the results of these tests in Figure 3.

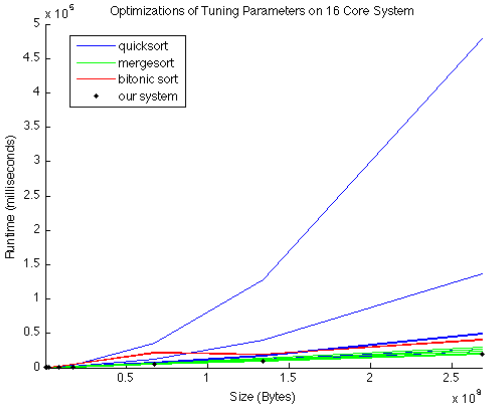

Figure 3: Performance on Machine Type 1

Figure 3 has various lines, consisting of three dif-

	Machine Type 1	Machine Type 2
Processor	AMD Opteron 6128 @ 2.0 GHz	Intel Core i5-3470S @ 2.9 GHz
Number of Cores	16	4
L2 Cache Size	512 KB	6144 KB
GPU	NVIDIA Quadro 600	NVIDIA GeForce GTX 660M
GPU Size	1047744 KB	523968 KB

Table 1: Specifications of machines used in tests

ferent colors. Each line represents a unique parameter for a given sorting algorithm. For example, the top blue line in Figure 3 represents running our system using a single threaded quick sort not over the network. Toward the bottom of Figure 3 there are many black dots. These black dots represent the run-time of our system using the parameters decided from the parameter search. For all of the red, blue, and green lines in Figure 3, the parameter search was not used, as the parameters were explicitly chosen from the start. However, the parameter search is included in the run-time when the system sorted the data using the parameters obtained from performing the parameter search.

In general, we found that the overhead due to the parameter search was quite small when compared to the actual time required to sort the data. On average, the parameter search took 50 milliseconds. This overhead may increase if the data file search space increases due to additional training on machines and data sets that are not accounted for in the data file set. Overall, we feel, even if the data file space increases, the overhead will never be overwhelmingly large because the system scans through a list of the file names that have yet to be filtered out, as opposed to continually performing I/O reads on the data files themselves.

By performing the parameter search, our system picked the best parameter combinations for this machine. However, the parameters chosen did vary depending on the input selected. After various tests, the main input characteristic that affected the choice of parameters was the size of the input.

As to parameters chosen by the parameter search in Figure 3, they are listed in Table 2. By observing the parameters chosen by the parameter search, in Table 2, for relatively small data sizes, the parameters chosen vary considerably among different data sizes. As Figure 3 shows, the parameter run-time curves are all very close together, meaning that the best choice in parameters up to 1 MB is not essential. In reality, the best parameter combination is around 10 milliseconds faster than the next best option. So, even if the best parameter is chosen, there is a chance another set of parameters can beat it due to noise in the system. In this range, there really is only a noticable difference when using the bitonic sort and the single threaded quick sort, but our system avoids those by performing the parameter search.

In the 64 MB to 512 MB range from Table 2, the best parameters converge to the eight-threaded merge sort not over the network. This is evident as well in Figure 3 because as the data size increases, one parameter combination, the eight-threaded merge sort not over the network, starts to separate itself from the rest of the parameter combinations in terms of lowest run-time.

In this test run, the input data's standard deviation was 2888. We did the same analysis as in Figure 3 and Table 2 for input data with a standard deviation of 10, 100, 1000, and 100,000. The results from those test were nearly the same as the results from the test when the standard deviation was 2888. For smaller input data sizes, the best parameter chosen varied. Then, for the large input data sizes, the best parameters converged to the eight-thread merge sort not over the network. These results were not unexpected. In Section 1.2, we reference "A Dynamically Tuned Sorting Library" [4] where the authors focus on how standard deviations affect sorting run-time. They choose algorithms, like radix sort, and tuning parameters that are heavily affected more by standard deviation. In our system, we do not include most of the parameters that are heavily influenced by standard deviation. The reason we included standard deviation into our auto-tuning system was to see if it had any affect on our results as it did with the referenced paper. Based on the results, the standard

Size of Input	Machine Type	Algorithm	Threads	Network?	run-time (ms)
16 KB	1	merge sort	8	no	248
128 KB	1	merge sort	2	no	253
1 MB	1	quick sort	8	no	339
16 MB	1	quick sort	4	yes	1461
64 MB	1	merge sort	8	no	5081
128 MB	1	merge sort	8	no	9996
512 MB	1	merge sort	8	no	20226
16 KB	2	merge sort	2	no	143
128 KB	2	merge sort	8	no	161
1 MB	2	quick sort	4	no	163
16 MB	2	merge sort	8	no	896
64 MB	2	merge sort	4	no	2814
128 MB	2	merge sort	4	yes	5833
512 MB	2	merge sort	4	yes	11349

Table 2: Parameters chosen based on type of machine and varying the input size

deviation has no affect on our tuning parameters.

Another noticeable discrepancy is that the bitonic sort is never chosen by our empirical search method for the tests we ran, as shown in Table 2 and is considerably slower than the best parameter for all data sizes, as displayed in 3. The main reason that this occurs is that our input sizes are not large enough to reap the benefits from using the GPU. The GPU has relatively large overhead when reading and writing to and from the GPU. So, if not enough data is used, the overhead of the GPU will trump the speedup gained from using the GPU. A similiar discrepancy can be seen in the network parameter, where hardly any of the best parameters chosen used the network. It is similiar in the sense that performing computation over the network has a considerable amount of overhead because communication and synchronization is required. As we point out in our future work section, our system's scalability needs to be tested with larger input data sets, where the benefits from using the network will override the overhead and potentially be used as a part of the best parameter combination for large data sizes.

In the second test run, we ran our system on a machine that fell under the category of Machine Type 2 from Table 1 using data sets varying in size but all having a standard deviation of 2888. The results of the tests are displayed in Figure 4.

Figure 4 is the exact same as Figure 3 in terms of format. Each colored line represents a unique parameter combination using a particular algorithm. The black dots represent the run-time of the system when it used the parameter search to choose the parameters. The choices of parameters are listed in Table 2. Based on the the parameters from Table 2 and the curves from Figure 4, the results have a substantial amount of similarities to the results from the first type of machines. For example, in Figure 4 most of the parameter combinations have about the same performance for sizes less than one megabyte. This is evident in Table 2 where the best parameters chosen by the parameter search varies a substantial amount. The other main similarity

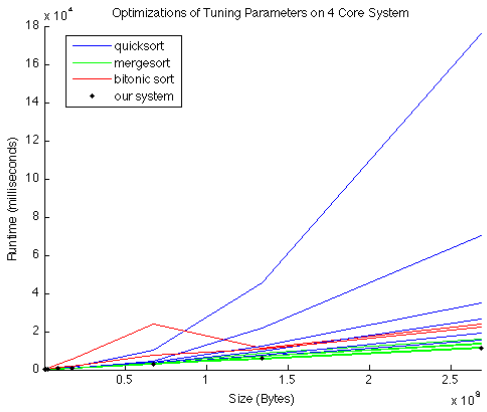

Figure 4: Performance on Machine Type 2

is that as the size of the input data increases, one parameter combination starts to win out.

For this system, the best parameter combination for large data sets was the four-thread merge sort over the network. This is a particulary significant result for a few reasons. First, the system did not choose the same parameter combination as the first system for large data sizes. The only parameter that the two had in common was the choice of the merge sort algorithm. Secondly, the system converged to the choice of using the network when sorting large sets of data. This means, for this particular type of machine, we use large enough sets of data to overcome the overhead from the network and see the speedup from using the network. Thus, it validates the inclusion of the network as a parameter option. Finally, notice that the optimal number of threads that the system converged on was four. This system has four cores. So, it makes sense that the machine performs best when there are four threads because any more threads cause additional overhead. Similarly, for the system in the first test, it converged toward the use of eight threads for large amounts of data. This makes sense as well becasue that system has sixteen cores. So, ideally, it will try to use as many threads as possible till all sixteen cores are used. In our system, we capped the number of threads to eight. So, appropriately, our library chose the eight-thread sort for the first system because it is the most amount of threads that does not exceed the number of physical cores on the machine.

As to the payoff from using the system to pick a set of parameters as opposed to picking an arbitrary set of parameters to use for sorting, here are a few examples. Assuming the data set is large, say 512 MB, if we picked the best parameter combination from the sixteen core machine and used it to sort the same data on the four core machine, the system performance is about 20% worse than the best parameter combination. For the reverse scenario, where the optimal parameters for the four core machine were used for the sixteen core, the performance is about 17% worse. For this data size, the performance overhead is small in terms of run-time as both of these percents translate to a two second slowdown. However, as the data sizes get larger and larger, this overhead could translate to minutes or even hours. In the worst case scenario, if the worst possible parameter combination was arbitrarily picked, the performance worsens by about twenty fold for both systems when compared to the best possible parameters.

3.3 Comparison to qsort()

In order to get a sense of how well our system performs compared to other sorting algorithms, we ran our system against C's stdlib qsort() function. The qsort() function is an in-place, sequential quick sort implemented in C's standard library. Therefore, there are no overheads due to using threads, the network, or the GPU. Figure 5 shows the performance of our system, which is running on the parameters chosen by the parameter search, against qsort().

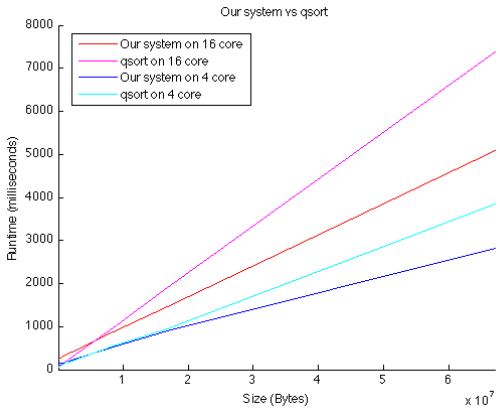

Figure 5: Performance compared to C's stdlib qsort()

Based on Figure 5, a few observations stand out. For both machine types, qsort() is faster than our system for relatively small input data, which is about 10 MB. The main reason as to why qsort() is faster for these smaller input sizes is that our system has a large overhead due to the fact that it uses the network, threads, and GPU. For example, the system is slowed down by factors such as the synchronization of threads, the synchronization of nodes in the network, communication between nodes, setup costs, and teardown costs. One example of these costs is that our system always calculates the standard deviation of the input data set.

However, as the size of the input increases, our system wins out and outperforms qsort() because

the parallelism of our system beats its related overhead. Based on Figure 5, qsort() performs about 42% worse on the sixteen core machine than our system on the sixteen core machine for a data input of size 512 MB. For the four core machine and an input size of 512 MB, qsort() performs about 37% worse than our system. As the data sizes increases, it is expected that our system will perform increasingly better than qsort() because the performance benefits of using the network or the GPU will increasingly outweight their associated overheads.

4 Future Work

Our results indicate that our auto-tuned library runs faster than C's stdlib qsort() for input data sizes larger than 10 MB. Our results also indicate that the speedup our system achieves in comparison to qsort() increases slightly as the input data size increases. Our future work includes gathering test results for input data sizes larger than 0.5 GB. We hypothesize that our sorting library, mainly due to its use of the network and GPUs, would significantly outperform sequential sorting algorithms on data sets larger than a single node's main memory.

Our results could be further strengthened by running experiments on more varied systems, too. Our results suggest that our auto-tuning framework correctly picks the most efficient algorithms for the two different types of machines we used in our experiments, but having more types of hardware would separate our system even more from a standard sorting function or library. The basic adaptability of our system, though, is demonstrated in Table 2.

Another source of potential improvement to our library would be the inclusion of more sorting algorithms. We chose to implement quick sort and merge sort in pthreads because of their different memory access patterns, but there exist more cache-conscious sorting algorithms that could be plugged in to our system, as well as more high performance sorting algorithms for GPUs. Overall, though, the disparity between quick sort and merge sort shows in our results, demonstrating the complexity of modern system's memory hierarchies.

5 Conclusion

In this paper, we introduced our auto-tuning sorting library for parallel sorting slgorithms. More specifically, our system tunes parameters such as which algorithm to sort with and the level of parallelism by deciding the number of threads to use, whether to sort over the network, or to use the GPU. The system makes the decision based on characteristics of the data such as input size and standard deviation and characteristics of the machine, which includes the number of cores, the cache size, and the GPU size.

The main contribution of this paper is that it shows how auto-tuning can significantly improve the performance of parallelized sorting algorithms running on complex systems. These types of algorithms are advantageous because they can sort large quantities of data much faster than sequential sorting algorithms. However, there are various parameters that must be decided upon when using such parallel algorithms. The wrong choice of parameters can lead to undesirable slowdown due to underutilization of hardware, overutilization of hardware, or overhead from using a particular piece of hardware. By using auto-tuning to pick these parameters, we are able to avoid these potential slowdowns by selecting the best parameters given the machine specifications and data characteristics.

Our system accomplishes the goal of being able to sort large data quickly and adapt based on the machine's specifications and the input data set's characteristics. Using a 512 MB input file, our system sorts large data quickly as it sorts 37% to 42% faster than C's sequential quick sort function. It is also able to adapt to machine specifications. For a 512 MB input file, it picked different parameter combinations for two different machines, one four core and the other a sixteen core machine. When the optimal parameter combination for the sixteen core system was used for the four core system, there was a 20% slowdown. Then when the optimal parameter combination for the four core system was used for the sixteen core system, there was a 17% slowdown. This shows that there is a significant payoff for choosing parameters that perform the best on a specific machine.

References

[1] David Cassell. A sort of a mess — sorting large datasets on multiple keys. pages 1–3, Corvallis, Oregon. SAS Institute, Inc.

[2] Jeffery Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*. Google, Inc., 2004.

[3] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[4] Xiaoming Li, María Jesús Garzarán, and David Padua. A dynamically tuned sorting library. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 111–, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Xiaoming Li, Maria Jesus Garzaran, and David Padua. Optimizing sorting with genetic algorithms. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 99–110, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with cuda based on bitonic sort. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, PPAM'09, pages 403–410, Berlin, Heidelberg, 2010. Springer-Verlag.

The Importance Of Discretization For Detecting Novel Attacks In Network Intrusion Detection

Christopher [Lekas, Magnano]
Swarthmore College
[clekas1,cmagnan1]@swarthmore.edu

Abstract

Detecting malicious network traffic is a costly problem in modern computing. Machine learning has been an active area of research for detecting malicious network activity. Many of the best performing detection systems require discretization of continuous data. Little work has been done to evaluate how discretization affects the performance of network intrusion detection systems. In this paper, a knowledge based discretization method based on the expectation-maximization algorithm is compared against a standard discretization method, the gain ratio. Both of these discretization methods are used in conjunction with random forests, a widely used classification algorithm. Results indicate that discretization methods can have a substantial impact on a network intrusion detector's ability to classify network activity, especially novel network attacks. Future work is needed to determine the exact strenghts and weaknesses of various discretization methods.

1 Introduction

As computer networks evolve, attacks and exploits of networks are becoming more complex. Network security is a large problem in modern computing. Intrusions of networks cost an estimated $100 billion per year(Lewis and Baker, 2013). While much of network defense is an arms race, where exploitations are found and removed as researchers and malicious users discover them, a critical task in network security is to detect malicious traffic in real time, even if those attacks are unknown. Many networks now incorporate a **network intrusion detector** (NID), which performs this task of detecting malicious traffic. Originally, many of these NID systems were able to recognize known attacks through expert systems(Shaker Ashoor and Gore, 2011); however, there has been a recent movement to use machine learning to detect network intrusions. The ultimate goal of a NID which uses machine learning is to be able to detect malicious traffic by learning general properties of network attacks and applying this knowledge to novel types of network attacks.

Many successful systems rely on finding splitting points in the data and often cannot natively handle continuous values. One problem with these methods is determining which ranges of values are important. This leads to the problem of **discretization**: how to split continues data up into a set of nominal values for the purposes of analyzing data for classification.

Discretization can have a large effect on the outcome of a classification and is especially important for methods, such as decision trees(Robnik-Sikonja, 2004), which cannot handle continuous features. Currently, most popular methods of discretization such as the **gain ratio** (GR) only allow linear separations of data through thresholding. See section 2 for an analysis of the GR. One weakness of thresholding is that the optimal choice of value ranges for classification may require a non-linear partition of

the data. This is especially true for many types of network attacks, which may require very specific ranges for length, time to live, etc. A classifier with only threshold discretization requires more steps to represent a non-linear partition of the data.

In order to guarantee that the ideal data partition is found for a given decision, all possible data ranges must be considered. However, this quickly becomes computationally infeasible. Some method must be used to select which ranges of values are analyzed when partitioning data for a given decision.

This paper presents **clustering based** discretization of data and compares it with the GR in order to determine the impact of discretization in network intrusion detection. Though discretization must lead to some level of information loss, incorporating the output of clustering into the discretization system can also add information to the data set. In this paper, a discretization method using unsupervised learning based on **Expectation Maximization** is explored.

The rest of the paper is laid out as follows: in Section 2, the unsupervised learning method for discretization and the main learning framework for the proposed NID are outlined. In Section 2.5, an explanation of how these methods are incorporated into the main NID framework is presented. Experimental methodology and data sets are presented in Section 3, results are examined in Section 4, and findings are summarized and future directions explored in Section 5.

2 Background

2.1 Related Work

A wide variety of machine learning methods have been applied to the problem of intrusion detection. Network intrusion detection presents a number of unique challenges for typical machine learning frameworks(Garcia-Teodoro et al., 2009). More popular methods include fuzzy logic, decisions trees, unsupervised learning, artificial neural networks, and artificial immune systems(Hofmeyr and Forrest, 2000). For a review of attempted methods see (Garcia-Teodoro et al., 2009).

The effect of discretization has been previously analyzed for decision tree based methods(Robnik-Sikonja, 2004). Other research has attempted to optimize currently used discretization methods(Quinlan, 1996); however, these analyses do not include clustering methods such as the proposed unsupervised learning method. To the authors' knowledge, this is the first analysis of discretization methods specifically for the problem of network intrusion detection.

2.2 Expectation Maximization

Expectation maximization (EM) is a widely used framework for the unsupervised learning task of clustering (Dempster et al., 1977). EM accomplishes this task by finding an maximum a posteriori (MAP) estimate of a set of hidden labels, Y, for a set of observed data, X, through an iterative two step process. This process consists of the expectation stage and the maximization stage. Each label in Y is associated with a multi-variate Gaussian function. These Gaussians are typically initialized to random values or are uniformly distributed over the range of values in X.

In the **expectation** stage, each data point x in X is assigned a level of membership to each label (ie cluster) y in Y. This membership level, l, is calculated as:

$$l_{xy} = \frac{e^{-(x-\mu_y)^2/(2\sigma^2)}}{\sum_{n=1}^{k} e^{-(x-\mu_n)^2/(2\sigma^2)}}$$

where σ^2 is the covariance of the set X, μ_y is the expected value of cluster y, and there are k total clusters. This is calculating the *expected* membership of x in y.

In the **maximization** stage, each Gaussian function associated with Y is altered so that it *maximizes* the probability of its set of membership levels. This new expected value for each cluster y in Y is calculated through

$$\mu_y = \frac{\sum_{x=1}^{j}(x \times l_{xy})}{j}$$

where j is the number of data points in X. This value can be viewed as the weighted average of all members of y. The expectation and maximization steps are repeated until the total MAP estimate crosses a certain threshold or a maximum number of iterations is reached.

The number of Gaussians or clusters used is fixed for a given run of EM. Typically, the number of Gaussians is chosen through performing EM on different numbers of Gaussians and choosing the number with the highest final MAP estimate.

2.3 Gain Ratio

The gain ratio (GR) is a calculation for discretization used in the C4.5 algorithm, one of the most commonly used algorithms for generating decision trees(Quinlan, 1993). For each feature, all possible splitting points are examined in the range of values within the data set. For each splitting point, the feature is treated as a nominal feature with two values: above the splitting point and below the splitting point. The GR is then calculated as the ratio of information gained by using the current partition to the intrinsic value of the partitioned data. More formally, information gain is defined as

$$InfoGain_F(D) = H(D) - \sum_{k=1}^{n} \left(\frac{|D_k|}{|D|} \times H(D_k) \right)$$

where D is a partition of the data, D_k is the subset of the partition where the target variable F is equal to k, n is the number of possible values of the F, and $H(X)$ is the entropy of the set X. The intrinsic value of a set is

$$IntrinsicValue_F(D) = -\sum_{k=1}^{n} \frac{|D_k|}{|D|} * \log_2 \left(\frac{|D_k|}{|D|} \right)$$

The gain ratio can then be calculated as

$$\frac{InfoGain_F(D)}{IntrinsicValue_F(D)}$$

The GR penalizes using features with large sets of unique values, which is not true of information gain alone. This has been shown to reduce over-fitting (Quinlan, 1993).

2.4 Random Forests

Random forests is an *ensemble* learning method based on decision trees. It generally outperform single decision trees(Breiman, 2001), and is less susceptible to over-fitting.

The decision tree model is a classic classification tool used in machine learning. Figure 1 shows and example of a decision tree for determining which drug to give a patient for a certain disease. Each internal node in the tree represents a condition or binary decision, and each leaf represents a classification or output. In figure 1 a green (thicker) line represents a true response to the parent node's condition, whereas a red (thinner) line represents a false response.

Random forests can be seen as an application and extension of bootstrap aggregating (bagging). Bagging is a general optimization of the supervised learning task of classification. Instead of a single strong classifier, a collection of weaker classifiers are created on subsets of the training data. The outputs of these classifiers are combined to create the final classification. Random forests extends this concept by creating each classifier in the ensemble using only a subset of the features of the total data set.

Figure 2 shows an example of a random forest. Random forests have been found to outperform single decision trees in many scenarios(Breiman, 2001). By training on random subsets of the total features, random forests can avoid over-fitting, one of the drawbacks of decision trees.

The implementation of random forests used constructs decision trees using the C4.5 algorithm (Quinlan, 1993). C4.5 uses the GR for discretization of continuous values, though data can also be discretized beforehand. For more information on C4.5, see (Quinlan, 1993).

2.5 The Intrusion Detection Pipeline

The proposed framework addresses the following classification problem: **Given:** Features of a packet received by a server such as packet length or protocol. **Output:** Classification of the given network packet as normal or as an attack, or as a specific attack group or attack type.

When enabled, EM is used to discretize all continuous features in the data set. One-dimensional EM is performed for each continuous feature separately, so each feature does not affect the discretization of any other feature. Once EM assigns labels to each continuous value, these labels are used as nominal values for the feature. When EM is not enabled, GR is used for discretization. Figure 3 summarizes the proposed NID framework.

Only training data was used for discretization. As

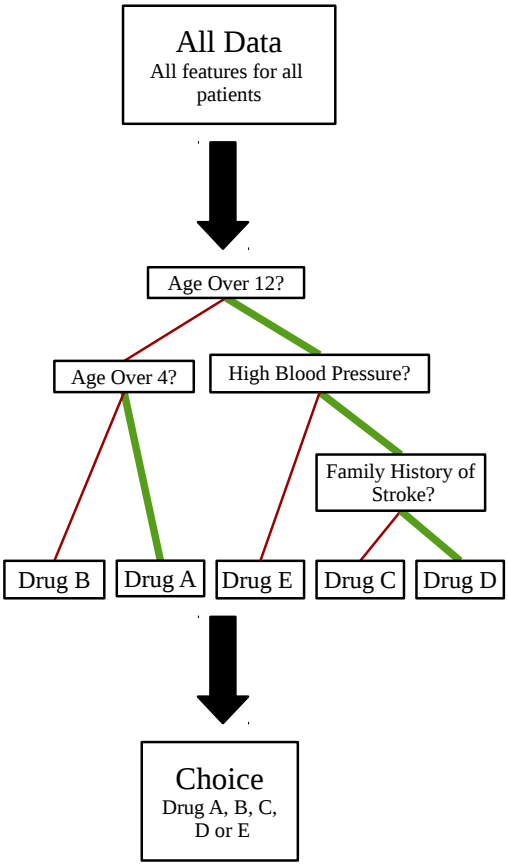

Figure 1: An example of a decision tree which determines which drug to give a patient for a disease.

EM is unsupervised learning, a NID could use all of the data for discretization without invalidating the results. However, only using training data better reflects the standard use case of a NID, in which retraining is infeasible due to the necessity of analyzing traffic in real time. Therefore, for real world NIDs, the current discretization would be based only on previously seen packets. The nominal features outputted by the discretization process were incorporated back into the data set, replacing the continuous features. Classification was then performed using random forests as outlined in the above section.

3 Experimental Methods

For analysis, 43 features were used. The definitions used for these features are outlined on the KDD99 description website(KDD99, 1999b). Implementation was done in java using WEKA(Hall et al., 2009) combined with an optimization of the standard WEKA random forest implementation, fast random forest(FastRandomForest, 2013). Other than the settings stated explicitly default settings were used.

All experiments were run on an iMac running Ubuntu 12.04 with an Intel Core i5 3470S CPU clocked at 1600MHz and 32GB of RAM.

3.1 Data Sets

As network intrusion detection is a popular field, there is a substantial selection of training and testing data available. The most widely used data set is referred to as "KDD99" or just "KDD" (Eldos et al., 2012) and was used in The Third International Knowledge Discovery and Data Mining Tools Competition, which was held together with KDD-99, The Fifth International Conference on Knowledge Discovery and Data Mining (KDD99, 1999b).

The Third International Knowledge Discovery and Data Mining Tools Competition used the cost matrix in Table 1 for scoring, so all costs computed

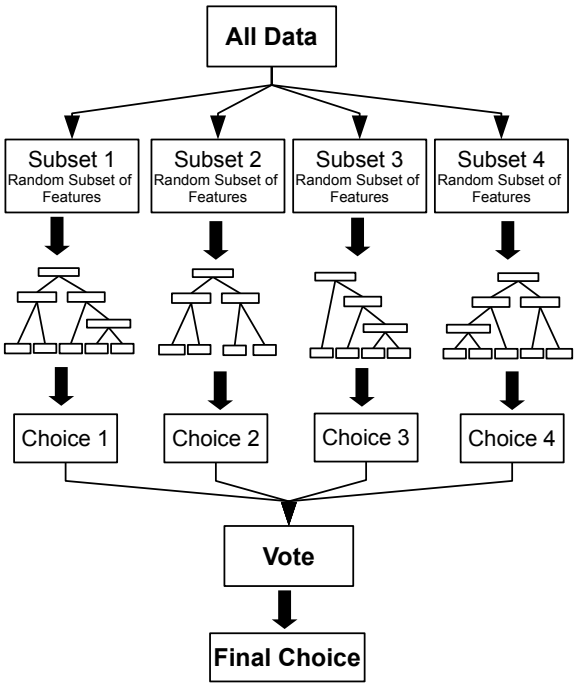

Figure 2: The random forest framework. An ensemble of weak classifiers is used as opposed to a single strong classifier.

	Normal	Probe	DoS	U2R	R2L
Normal	0	1	2	2	2
Probe	1	0	2	2	2
DoS	2	1	0	2	2
U2R	3	2	2	0	2
R2L	4	2	2	2	0

Table 1: Cost matrix used in The Third International Knowledge Discovery and Data Mining Tools Competition, as well as many subsequent papers. Columns correspond to groups packets were classified as; rows correspond to groups that packets were actually members of.

for this paper used the same cost matrix.

3.1.1 KDD

KDD includes packets tagged with a variety of attack types, each of which is a member of one of five packet groups: normal, probe, DoS (denial of service), U2R (user to root), or R2L(remote to local). Aside from a label, the data for each packet are limited to information that would be readily available either at the interface between the program being protected by the NID and the network or in the packet itself, such as protocol, number of failed logins, and various flags. It contains 4,898,431 observations.

KDD results were included to provide a benchmark for comparison with other methods. However there are several known problems with the KDD data set (Tavallaee et al., 2009), such as different distributions of attack types in the training and testing sets.

For particularly time-intensive algorithms, the amount of data needing to be processed is prohibitive. EM is one such time-intensive algorithm. Although five folds of the RF portion of the classification process can be completed in under one hour when run on the full KDD training set (using the

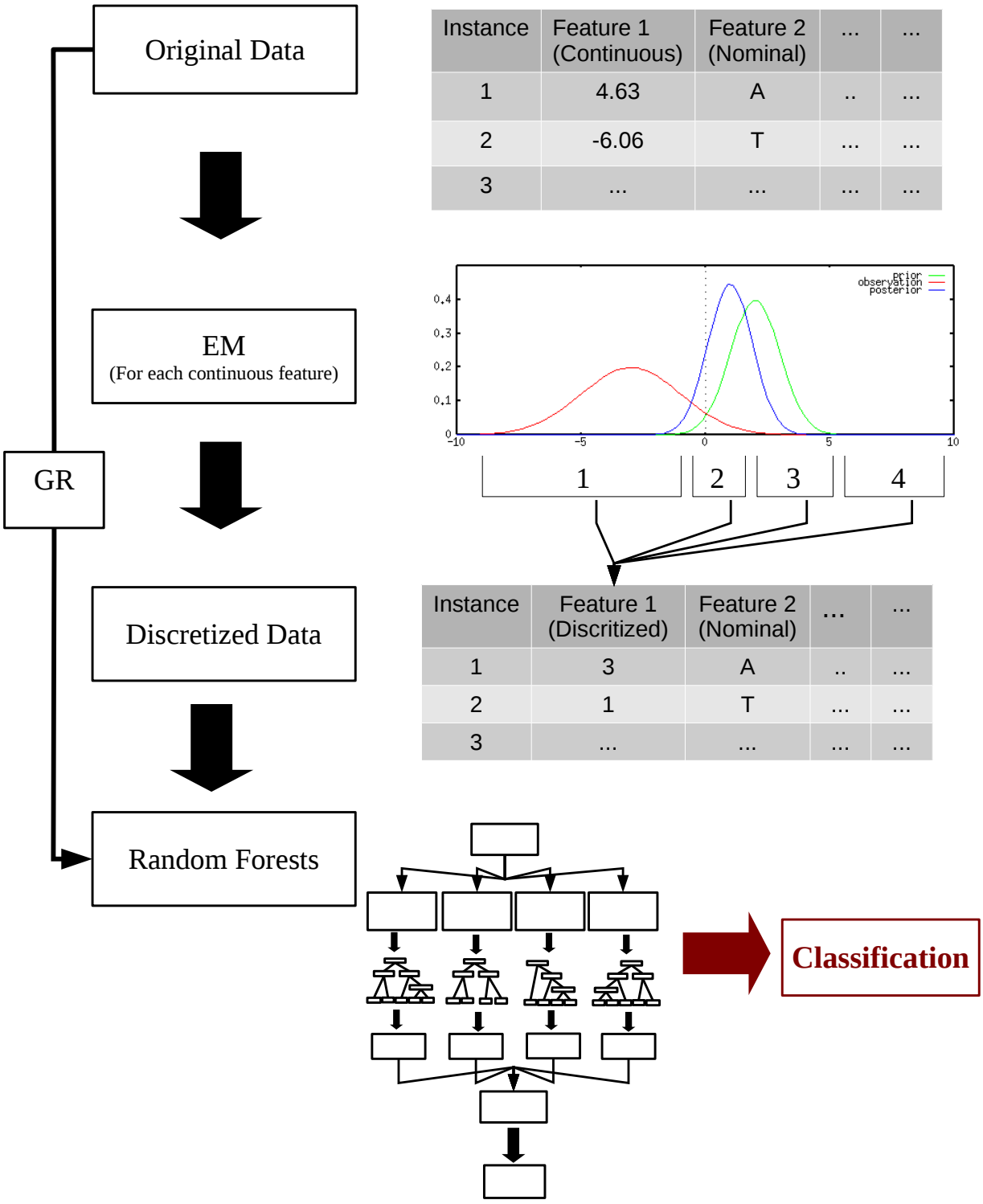

Figure 3: The proposed intrusion detection pipeline. Note that the EM steps can be skipped, in which case GR is automatically performed by the C4.5 algorithm

FastRandomForest modification of the Weka RandomForest class with no additional modifications for efficiency), adding EM to the program causes the estimated runtime of five folds to be more than 60 days.

3.1.2 NSL-KDD

The NSL-KDD dataset is an updated version of KDD in which many of the problems present in the original KDD data set are fixed(Tavallaee et al., 2009). Whereas KDD contains 4,898,431 observations, NSL-KDD only contains 125,974. Because of its improvements over KDD, NSL-KDD is used as the primary data set for experiments. All of the experiments run on NSL-KDD were also run on KDD. KDD results are provided in appendix E for comparison to official competition results (KDD99, 1999a).

3.2 Experiments

To guide the development of experiments, "stories" were developed describing important, general scenarios that network intrusion detectors were likely to encounter. For each experiment, the

Four experiments were developed, three of which were based around what were decided to be the most fundamental and crucial stories:

- NID encounters a known attack type.
- NID encounters a novel attack type.
- NID encounters a novel attack group.

The fourth experiment was performed using the data used in the KDD'99 Classifier Learning Contest and the equivalent NSL-KDD data. The purpose of this experiment was primarily to produce results that could be compared with those of other network intrusion detectors.

Due to the large size of the KDD data set, a 1% subset of the total KDD data set was used to determine the number of clusters for each feature on all experiments in which EM was run on the full KDD data set, in order to reduce run time. Between the Gaussian estimation and other memory optimizations, the program with EM enabled was able to run five-fold cross-validation on the full KDD training set in under 11 hours, a 130x speedup.

3.2.1 Known Attack Types Experiment

To assess how the two systems would perform when classifying only known attack types, each was tested separately on the full NSL-KDD data set. 5-fold cross-validation was used to enable both training and testing on all available data, ensuring that any strange observations were able to both alter the model and be classified by it.

3.2.2 Novel Attack Types Experiment

The story behind this experiment is that a NID encounters an attack type that it has not trained on. As NIDs are generally fairly successful at classifying known attacks, aggressors consistently output new attacks in hopes of beating NIDs.

In order to test how well a network intrusion detection system could classify a novel attack type, the same data sets were used as for classifying known attack types. For each run, a different random fifth of the data set was removed to decrease over-fitting. From the remaining four fifths of the data, one attack type was used as the testing data and the rest of the data were used for training. All of the novel attacks types were chosen from R2L because of R2L's large variation in attack characteristics (Sabhnani and Serpen, 2003) and so that the knowledge that was trained on would be approximately the same for each attack. The disadvantage of choosing R2L is that, even with improvements made in NSL-KDD, there are still very disproportionate numbers of attacks of each type. R2L's affliction lies in warezclient making up the bulk of its observations. Unfortunately, all of the other attack groups have characteristics that make them less desirable choices for this experiment than R2L. In the case of U2R, there are not enough observations for it to be used. Probe and DoS both have large numbers of observations, but have similar disproportionate distributions of observations amongst attack types to U2R, as well as less variety in attacks and enough observations to drastically alter the model by excluding one of their attack types from training.

The question to which an answer was desired was "What if a network intrusion detector has been trained on all existing attack types and a new attack type appears?", not "What if the training set does not include one of the currently existing attack types?", where an existing attack type is any attack

type that had ever been used anywhere before the time the system was trained. Although the difference between these two questions may seem to be nothing more than a bit of wordplay, the latter has the potential to bias results.

The latter case decreases the quality of the training set by removing information about a given attack group (the source of potential bias, as the excluded data would be used for training any NID that were to be used for real-world applications), causing observations of known attack types to potentially be misclassified more often than they would have otherwise been. Additionally, the poorer quality training set would be expected to cause any novel attack type to be misclassified more often because the model would have less knowledge about where in feature space the boundaries of the attack groups lay. The case of a training set not including a known attack is also less important than that of a new attack showing up because the former can be solved by gathering more training data.

The existence of a new attack does not affect the quality of any existing model until the model learns from it because any classification algorithm will always classify a given observation in the same way, unless the classifier's underlying model changes. The catch in this question is that any attack that can be used for training or testing purposes is an existing attack, so the experiments carried out must technically be answering the second question.

Answering the wrong question, in this case, turns out to not be a major problem. It is acceptable to pretend that all existing attack types were trained on in each novel attack type test (i.e. that the novel attack is unseen) because the potential bias affects both discretization methods. Since the main question is how the two methods perform relative to each other, this approximation is safe.

Additionally, in these tests, it does not matter how well the systems perform on known attack types, as the tests for known attack types were run separately. In any real-world NID, all of the available data would be used for training, so the impact of the exclusion of data on the classification of known attack types does not provide any meaningful information. Therefore, measurements of precision and F-score also lack meaning for these tests. It is important to note that comparisons between leave-out experiments should be made cautiously as leaving out an attack type or group creates slightly different data distributions.

3.2.3 Novel Attack Groups Experiment

The story for this experiment is that a NID encounters an attack group that it has not trained on. The essential difference between a novel attack type and a novel attack group is that the latter is indicative of an entirely new method of attack. This experiment was not carried out to answer the question of how a classifier would perform if all of the attack types in a group that the classifier had not been trained on were suddenly observed, but to determine how the classifier would fare against a single attack type of an unknown group.

To test the classifiers' ability to detect attacks from unknown groups, NSL was used with one attack group removed from the training data and all but that one attack group removed from the testing data. The results obtained from this experiment were therefore aggregated the attacks by group, giving a more intuitive sense of how easy it was to classify some representative new attack type from a new group of attacks.

3.2.4 KDD99 Competition Experiment

Many of the results available from academic network intrusion detection systems are provided as cost matrices and average costs from training and testing on the official KDD data. In order to provide results that can be compared to those, training and testing were performed using the the official KDD data.

4 Results And Discussion

As NSL-KDD was believed to provide training and testing data that were more useful for learning about and testing a NID's ability ability to handle real world attacks, respectively, it was chosen as the data set to analyze the results of. Despite this belief about NSL-KDD's preferability to KDD, tests were run on KDD as well in order to increase comparability of the NIDs described in this paper with other network intrusion detection systems. The results for both tests on KDD and tests on NSL-KDD are provided; however, only the results of the tests on NSL are discussed.

Discretization Strategy	Average Cost	Precision	Recall	F-Score
EM	0.00337	0.99845	0.99828	0.99836
GR	0.00338	0.99848	0.99824	0.99836

Table 2: KDD-NSL average costs, precisions, recalls, and F-scores when testing on known attack types. For precision, recall, and F-score, any attack is positive and normal is negative.

4.1 Experiment Results

4.1.1 Known Attack Types Results

Table 2 shows that, for both EM and GR, average cost is close to zero, while precision, recall, and F-Score are all nearly one. These measures signify that both discretization strategies are highly effective for constructing models that will be tested on known data. Furthermore, the maximum difference between the two discretization strategies for any of these measures is 0.00004, signifying that their effectivenesses are similar.

The precisions and recalls of the individual groups are very similar between the EM and GR versions of the RF model (See the confusion matrices in Appendix B). Of these precisions and recalls, only the ones for U2R fall below 95%. It is unsurprising for U2R to have such low precision and recall because there are so few U2R observations in the data set and, therefore, few U2R observations in the training data. As R2L attacks are generally the most varied, it seems that R2L should have the lowest precision and recall of the four groups, lending additional credence to the supposition that U2R's poor results are due primarily to lack of observations in the training data.

The results of this experiment support the those of prior research which found NIDs to be mostly successful at classifying known attack types, (Verwoerd and Huney, 2002) further validating this project's emphasis on the ability of NIDs to classify novel attacks.

4.1.2 Novel Attack Types Results

The ability of the NIDs to classify novel attack types is underwhelming, as evidenced in Table 3, especially in comparison to the the NIDs' results from the known attack types experiment. For four of the seven comparable novel attack types (i.e. not warezclient), neither NID successfully classified a single observation. For only two of the seven comparable attack types did both NIDs successfully classify at least one observation correctly. For KDD results see Appendix C.

Given the highly inequitable distribution of observations among attack types in the R2L group discussed in the experimental methods section, these results are perhaps not too surprising. R2L's distribution problem is likely to bias all recalls downward. When excluding warezclient from training, there are very few R2L observations to train on, so the model is unlikely to predict warezclient attacks correctly. For all of the other attack types, most of the R2L training has been done on warezclient attacks, so any attack type in the R2L group that is not particularly similar to warezclient is likely to be misclassified.

The difference between the recall of warezclient for EM and that for GR is very small. EM correctly classified two out of 890 warezclient observations correctly, which is only slightly more than GR's zero out of 890; however, zero is a particularly troublesome number because GR might have been about to classify instances correctly or it might have been making terrible errors in its classifications, but the data available cannot confirm which case it is. In the former case, the difference between the two recalls for warezclient would be considered to be negligible, whereas in the latter case it would be considered significant.

Among the seven comparable novel attack types, three have non-zero differences between the recalls for EM and GR. Again, the other four comparable attack types have no observations classified correctly by either system. All of the three non-zero differences have magnitudes greater than 0.1, which the authors of this paper had decided in advance to be a threshold above which a difference was to be treated as important. Three out of seven differences being important is evidence that choice of dis-

Novel Attack Type	GR	EM	Δ	# Obs
ftp_ write	0.31250	0.00000	-0.31250	32
guess_ passwd	0.00000	0.00000	0.00000	212
imap	0.00000	0.00000	0.00000	44
multihop	0.50000	0.25000	-0.25000	28
phf	0.00000	0.00000	0.00000	16
spy	0.00000	0.00000	0.00000	8
warezclient	0.00000	0.00028	+0.00028	3560
warezmaster	0.48750	0.71250	+0.22500	80

Table 3: NSL-KDD recalls of novel attack types.

cretization strategy is important for classification of novel attack types; however, not all of these three differences favor the same discretization strategy: two are in the favor of GR, but one favors EM. Although different discretization strategies have varying strengths and weaknesses, the results imply that which strategy is chosen has important an important impact on classification of novel attack types. Further investigation is required to determine what the strengths and weaknesses of various strategies are.

Whether the differences in these results were caused spuriously is not clear because the numbers of observations are very low for all of the novel attack types. To obtain a better understanding of how meaningful these results are, they are checked against the novel attack group results for consistency in the following sections.

4.1.3 Novel Attack Groups Results

Although the recalls in Table 4 are much lower than those found in the known attack types experiment, they are all non-zero. With far more observations for all groups but U2R, the novel attack groups experiment provides more obviously significant evidence about the importance of discretization strategy choice. For KDD results see Appendix D. Probe, DoS, and U2R all have recall difference magnitudes greater than 0.1, supporting the case that there is an important difference in classification ability between EM and GR. DoS' recall's difference magnitude, in particular, is extremely large, being close to 0.6. It is difficult to come up with any way to explain away this particular difference between GR and EM, as its magnitude is large and DoS has over 180,000 observations. The difference in correct classification of probe by EM and GR is similarly significant due to the more than 45,000 probe observations. The significance of U2R's recall difference is unclear due to U2R's small magnitude.

R2L's recall difference is not important in the rather arbitrarily sense prescribed by this paper's authors; however, GR's recall for R2L is around nine times larger than EM's is. Even if this is determined to be an insignificant difference, there are still two groups for which there are important and significant differences between the recall for GR and that for EM. With two out of four groups having clearly significant and important differences, it is apparent that choice of discretization strategy is important in for classification of novel attack groups. The important results obtained in both novel attack group and attack type experiments which are based on few observations may also be significant.

Despite having two novel attack groups for which there are important and significant difference between the classification abilities of GR and EM, it is not clear whether EM or GR is generally a better discretization strategy. Since the differences for probe and DoS have different signs, it again appears that EM's and GR's strengths lie in different places when it comes to novel attack classification.

4.1.4 KDD99 Competition Results

Table 5 shows that the RF model with EM discretization classifies data such that the average cost of a classification is lower than for the RF model with GR discretization, although the average costs of both are close. Furthermore, their average costs are both extremely high. As the average cost in the cost matrix for probe is less than two, a NID

Novel Atk Group	GR	EM	Δ	# Obs
probe	0.29212	0.15462	-0.13750	46624
DoS	0.31440	0.89109	+0.57669	183708
U2R	0.17788	0.06250	-0.11538	208
R2L	0.02386	0.00276	-0.02110	3980

Table 4: NSL-KDD recalls of novel attack groups.

Discretization Strategy	Average Cost
EM	2.65374
GR	2.70320

Table 5: KDD competition data average costs for NSL-KDD.

which classified all packets as probes would have a substantially lower average cost than RF with either of the discretization strategies used in this paper. Since EM and GR performed almost equally poorly, despite having exhibited important differences in earlier tests, it seems likely that the large average costs are caused by some pathological problem in RF and/or KDD-NSL.

See Appendix F for the confusion matrices which produced the average costs found in Table 5.

4.2 Aggregating The Results

From the novel attack type and novel attack group experiments, it is apparent that discretization strategy is an important factor in a NID's ability to correctly classify incoming packets; however, it is not clear what an optimal discretization strategy is. EM and GR each performed especially well in different areas and neither clearly performed better on known packet types.

5 Conclusion and Future Work

It has been shown that discretization is an important factor in model selection for a NID. Though the proposed method of discretization, EM, does not consistently outperform more common discretization methods it has been shown that, in certain situations, the rate of detection can be significantly improved through choosing an approproate discretization strategy. One interesting future direction of research would be to analyze and determine in which situations EM discretization results in more accurate intrusion classification than the GR.

Discretization necessitates information loss. EM as implemented for this paper loses more information then GR does because it creates Gaussians for features independently of the target feature or other features. It is possible that a multidimensional implementation of EM would be more successful due to its ability to consider the relationships among continuous variables (i.e. its ability to retain more information). Perhaps this would allow EM to perform closer to GR in the areas in which GR was superior without sacrificing the advantages that EM demonstrated.

There is likely a better discretization strategy that has not been considered. Many other discretization strategies exist, such as Gini coefficient, MDL (maximum delineation length), ReliefF, and MyopicReliefF. (Garcia-Teodoro et al., 2009) Furthermore, hybrid strategies can be developed using techniques like voting systems. As EM and gain ratio each have areas in which they are definitively superior to each other, a hybrid strategy is a strong candidate for producing better discretization.

Another possible future direction would be using a more knowledge based discretization framework to incorporate domain knowledge into the machine learning model. Network administrators could use the kinds of traffic they expect for a given network to model the discretization scheme. This could allow a high degree of customization when deploying a given system. For instance, a network administra-

tor for a video streaming service could create a special IP discretization which parses by country of origin, allowing the machine learning system to be able to detect illegal video streaming traffic more easily. This could allow increased human readability and increased NID accuracy.

Recently, the area of adversarial machine learning has become a prominent topic in network intrusion detection. Adversarial machine learning involves a malicious user intentionally mis-training the NID in order to get malicious traffic through the NID undetected. For a more thorough review of adversarial machine learning, see (Huang et al., 2011). An interesting future direction would be to explore how discretization can affect the upper bound on how much a malicious user can alter the space of a classifier.

References

Leo Breiman. 2001. Random forests. *Machine Learning*, 45:5–32.

A.P. Dempster, N.M Laird, and D.B. Rubin. 1977. Maximim likelihood from incomplete data via the em algorithm. *Journal of the Royal Society, Series B*, 39(1):1–38.

Taisir Eldos, Mohammad Khubeb Siddiqui, and Aws Kanan. 2012. On the kdd'99 dtaset: Statistical analysis for feature selection. *Journal of Data Mining and Knowledge Discovery*, 3(3).

FastRandomForest. 2013. http://code.google.com/p/fast-random-forest/.

P. Garcia-Teodoro, J. Diaz-Verdejo, G. Marcia-Fernandez, and E. Vazquez. 2009. Anomaly-based network intrusion detection: Techniques, systems, and challenges. *Computers & Security*, 28:18–28.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The weka data mining software: an update. *SIGKDD Explorations*, 11(1).

Steven Hofmeyr and S. Forrest. 2000. Architecture for an artificial immune system. *MIT Press*, April.

Ling Huang, Anthony Joseph, Blaine Nelson, Benjamin Rubinstein, and J.D Tygar. 2011. Adversarial machine learning. In *Proceedings of the 4th ACM workship on Security and Artificial Intellegence*, pages 43–58.

KDD99. 1999a. http://cseweb.ucsd.edu/ elkan/clresults.html.

KDD99. 1999b. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

James Lewis and Stewart Baker. 2013. The economic impact of cybercrime and cyber espionage, July.

J.R. Quinlan. 1993. *C4.5: Programs for Machine Learning*. San Mateo: Morgan Kaufmann.

J.R. Quinlan. 1996. Improved use of continuous attributes in c4.5. *Journal of Artificial Intelligence Research*, 4:77–90.

Marko Robnik-Sikonja. 2004. Improving random forests. In *Proceedings of ECML, Springer, Berlin*.

Maheshkumar Sabhnani and Grsel Serpen. 2003. Kdd feature set complaint heuristic rules for r2l attack detection. In Hamid R. Arabnia and Youngsong Mun, editors, *Security and Management*, pages 310–316. CSREA Press.

Asmaa Shaker Ashoor and Sharad Gore. 2011. Importance of intrusion detection systems (ids). *International Journal of Scientific & Engineering Research*, 2(1), January.

Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali-A. Ghorbani. 2009. A detailed analysis of the kdd cup 99 data set. In *Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defence Applications*.

Theuns Verwoerd and Ray Huney. 2002. Intrusion detection techniques and approaches. *Computer Communications*, 25:1356–1365, September.

A Known Attack Types KDD Results

Discretization Strategy	Average Cost	Precision	Recall	F-Score
EM	0.01007	0.99752	0.99730	0.99741
GR	0.00010	0.99998	0.99997	0.99997

Table 6: KDD average costs, precisions, recalls, and F-scores when testing on known attack types. For precision, recall, and F-score, any attack is positive and normal is negative.

B Known Attack Types Confusion Matrices

	Normal	Probe	DoS	U2R	R2L	Recall
Normal	958603	1301	3217	14	40	0.99525
Probe	9250	26402	5043	0	1	0.64876
DoS	559	3	3843978	0	0	0.99985
U2R	24	0	0	25	1	0.50000
R2L	647	3	2	1	467	0.41696
Precision	0.98919	0.95283	0.99786	0.62500	0.91749	

Average Cost:	0.01007
Precision:	0.99752
Recall:	0.99730
F-score:	0.99741

Table 7: Confusion matrix for KDD with EM. Columns correspond to groups packets were classified as; rows correspond to groups that packets were actually members of. For precision, recall, and F-score, any attack is positive and normal is negative.

	Normal	Probe	DoS	U2R	R2L	Recall
Normal	972702	40	24	6	9	0.99992
Probe	52	41045	5	0	0	0.99861
DoS	20	3	3883345	0	2	0.99999
U2R	22	1	0	28	1	0.53846
R2L	20	0	2	3	1101	0.97780
Precision	0.99988	0.99893	0.99999	0.75676	0.98922	

Average Cost:	0.00010
Precision:	0.99998
Recall:	0.99997
F-score:	0.99997

Table 8: Confusion matrix for KDD with GR. Columns correspond to groups packets were classified as; rows correspond to groups that packets were actually members of. For precision, recall, and F-score, any attack is positive and normal is negative.

	Normal	Probe	DoS	U2R	R2L	Recall
Normal	67275	35	17	4	12	0.99899
Probe	41	11609	4	1	1	0.99597
DoS	17	9	45901	0	0	0.99944
U2R	16	3	0	31	2	0.59615
R2L	27	0	0	3	965	0.96985
Precision	0.99850	0.99597	0.99954	0.79487	0.98469	

Average Cost:	0.00337
Precision:	0.99845
Recall:	0.99828
F-score:	0.99836

Table 9: Confusion matrix for NSL-KDD with EM. Columns correspond to groups packets were classified as; rows correspond to groups that packets were actually members of. For precision, recall, and F-score, any attack is positive and normal is negative.

	Normal	Probe	DoS	U2R	R2L	Recall
Normal	67274	36	18	4	11	0.99897
Probe	39	11612	3	1	1	0.99623
DoS	19	8	45900	0	0	0.99941
U2R	18	2	0	30	2	0.57692
R2L	27	0	0	3	965	0.96985
Precision	0.99847	0.99605	0.99954	0.78947	0.98570	

Average Cost:	0.00338
Precision:	0.99848
Recall:	0.99824
F-score:	0.99836

Table 10: Confusion matrix for NSL-KDD with GR. Columns correspond to groups packets were classified as; rows correspond to groups that packets were actually members of. For precision, recall, and F-score, any attack is positive and normal is negative.

C Novel Attack Types KDD Results

Novel Attack Type	GR	EM	Δ	# Obs*
ftp_ write	0.28125	0.00000	-0.28125	32
guess_ passwd	0.00000	0.00000	0.00000	204
imap	0.00000	0.00000	0.00000	48
multihop	0.50000	0.35714	-0.14286	28
phf	0.12500	0.00000	-0.12500	16
spy	0.00000	0.00000	0.00000	8
warezclient	0.00049	0.00000	-0.00049	4064
warezmaster	0.65000	0.08750	-0.56250	80

*Number reported is for testing with EM. Because the implementation with EM was tested on only 99% of the full KDD data set, GR had 16 additional observations for warezclient (4080 total).

Table 11: KDD recalls of novel attack types.

D Novel Attack Groups KDD Results

Novel Atk Group	GR	EM	Δ	# Obs*
probe	0.40197	0.41542	+0.01345	162784
DoS	0.21118	0.23597	+0.02479	15378160
U2R	0.12500	0.12000	-0.00500	200
R2L	0.01465	0.01071	-0.00394	4480

*Numbers reported are for testing with EM. Because the implementation with EM was tested on only 99% of the full KDD data set, GR had additional observations for each group in the following amounts: 1624 for probe (164408 total), 155320 for DoS (15533480 total), 8 for U2R (208 total), 24 for R2L (4504 total).

Table 12: KDD recalls of novel attack groups.

E Competition KDD Results

Discretization Strategy	Average Cost
EM	0.29267
GR	0.27458

Table 13: KDD competition data average costs for KDD data.

F Competition Data Confusion Matrices

	Normal	Probe	DoS	U2R	R2L	Recall
Normal	59366	64	163	972	28	0.97975
Probe	194	27	0	4	3	0.11842
DoS	7751	490	221612	0	0	0.96415
U2R	16083	95	0	3	8	0.00019
R2L	1340	2510	313	0	3	0.00072
Precision	0.70062	0.00847	0.99786	0.00306	0.07143	
Average Cost:	0.29267					

Table 14: Confusion matrix for KDD with EM. Columns correspond to groups packets were classified as; rows correspond to groups that packets were actually members of.

	Normal	Probe	DoS	U2R	R2L	Recall
Normal	59788	302	503	0	0	0.98671
Probe	81	130	4	6	7	0.57018
DoS	5984	149	223623	0	97	0.97290
U2R	15486	522	124	1	56	0.00006
R2L	527	3472	166	0	1	0.00024
Precision	0.73032	0.02842	0.99645	0.14286	0.00621	
Average Cost:	0.27458					

Table 15: Confusion matrix for KDD with GR. Columns correspond to groups packets were classified as; rows correspond to groups that packets were actually members of.

	Normal	Probe	DoS	U2R	R2L	Recall
Normal	1651	2769	2772	0	266	0.22137
Probe	2662	82	5	2	3	0.02977
DoS	9037	266	401	5	2	0.04129
U2R	176	3	5	10	6	0.05000
R2L	1130	1275	16	0	0	0.00000
Precision	0.11265	0.01866	0.12535	0.00306	0.58824	
Average Cost:	2.65374					

Table 16: Confusion matrix for NSL-KDD with EM. Columns correspond to groups packets were classified as; rows correspond to groups that packets were actually members of.

	Normal	Probe	DoS	U2R	R2L	Recall
Normal	1474	2905	3079	0	0	0.19764
Probe	2563	180	3	3	5	0.06536
DoS	9415	214	82	0	0	0.00844
U2R	193	0	0	7	0	0.03500
R2L	707	1549	164	0	1	0.00041
Precision	0.10270	0.03713	0.02464	0.70000	0.16667	

Average Cost: 2.70320

Table 17: Confusion matrix for NSLKDD with GR. Columns correspond to groups packets were classified as; rows correspond to groups that packets were actually members of.

Determining Offloading Point of Image Processing Implementations: A Software versus Hardware Comparison

Danielle Sullivan
Swarthmore College
dsulliv2

Eliza Bailey
Swarthmore College
ebailey1

Taylor Nation
Swarthmore College
tnation1

December 20, 2013

Abstract

Field programmable gate arrays (FPGAs) are integrated circuits that give users the ability configure the hardware after the board has been manufactured. Due to their highly customizable nature, FPGAs can offer parallelization and program speedup that a CPUs computational ability typically prevents. This benefit does not come without drawbacks; the speed and latencies associated with using FPGAs is offset by the cost of transferring data between it and the CPU. Therefore, running smaller scale programs on the CPU may be more desirable than incurring the data transfer penalty versus utilizing CPU core cycles. The FPGA accelerated function may have higher latencies and completion times than the same Software based function or algorithm running on the CPU.

Our project finds the point at which running a hardware implementation on an FPGA is faster than the functional software version based on run time. In other words, we seek to find the estimated size of an input image at which the runtime of using the CPU to perform calculations is greater than the cost of exporting the data to the FPGA plus the runtime of the program in hardware. We developed a modeling application that takes an python image processing file, the degree of the polynomial that will fit the runtime test results, and an estimation of the percent speedup achieved through the FPGA; and then calculates the offloading point. This point being the size of a dataset at which it would be faster perform computations on the FPGA than on the CPU. The image processing code we tested our application on converted an image to black and white, and the hardware we tested with was the Altera DE2 board. Knowing this point is valuable because it enables the user to make an informed choice between the hardware or software version of their program based on their needs.

1 Introduction

It is well known that hardware offers speed up that not achievable by most software implementations of a program. One such piece of hardware used to achieve this speedup is the Field Programmable Gate Array (FPGA), which allows the engineer to program the hardware directly. The engineer is responsible for selecting inputs, processing methods, and output, giving the programmer a large amount of control over how the board is programmed and what it can do. In addition to (and as a result of) the customizable nature of the FPGA, it is incredibly conducive to parallel programming; the programmer can simply program different parts of the board to do different operations, and the result of the process is that it runs in parallel. Clearly, the FPGA has a lot of great applications, and is very useful to speeding up runtime for most programs.

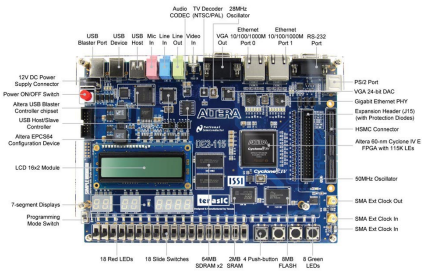

Figure 1: Layout of the DE2 Altera board

However, in spite of all of the benefits that FPGAs have, using them is not without some frustration. Since the board is not hardwired into the computer, offloading a computation to an FPGA incurs some overhead. The process includes moving the data from the CPU to the FPGA, giving control to the FPGA, performing the calculations, moving all of the data back onto the CPU, and then returning control of the

program to the CPU. When the data set is not sufficiently large, using the FPGA could possibly do more harm than good, due to its computational expense. However, there is a large enough set (often referred to as the offloading point) that using the FPGA is faster than using the CPU.

2 Motivation

Digital image processing (DIP) is a rapidly growing field with a wide range of applications from medicine to national defense. As the resolution and number of pixels of images grow, software implementation of DIPs grows increasingly less efficient; this is doubly true for real-time applications. Research has shown that offloading these techniques onto hardware, including FPGAs, creates significant speed up[6]. Therefore, finding the data size where it is faster to run DIPs on hardware rather than software is important.

FPGAs offer many advantages over CPUs, such as high performance, optimization, computational density, and reliability. In addition FPGAs are known for being highly reliable and having low costs. Finally FPGAs are truly parallel, making them very attractive for computationally expensive tasks which could easily slow down a computer[1].

3 Background

It is well known that some of the main considerations when choosing between a software and hardware implementation is weighing the features and flexibility of software against the superior speed and power consumption of hardware. [2]. Many projects are using a combination of software and hardware implementations (SW/HW) [5] to reap the benefits each offer in ways that result in the highest overall program performance. The difficult question to answer becomes, which parts of the overall system should be implemented in software and which in hardware. Currently, researchers look at variables such as parallelizability, frequency of use, and the hardware capabilities in order to make that decision [7]. Our projects uses variable data input sizes to analyze the runtime of one potential aspect of a program, in order to determine what size data results in enough of a slowdown of the software implementation that hardware becomes advisable.

FPGAs are re-configurable hardware that offer benefits such as the ability to be an efficient matrix multiplier in one run, and an image processing tool the next. How much speed up they offer is heavily dependent, as G. R. Morris coins, on "the two p's, pipelining and parallelism."[4] Their ability to offer speedup is best seen in highly parallelizable algorithms that can take advantage of the FPGAs parallel nature. Morris explains that programs that should be considered for mapping onto an FPGA rely on the "p heuristics," parallelization and pipelining. Other considerations include the memory bandwidth of the hardware, and the potential of the data reuse. If a single function offers 1000% speedup, but only constitutes 1% of the overall program, offloading may just not be worth it. Morris calls this decision the "FPGA Design Boundary", and our project seeks to determine this automatically, using the metric of program runtime. Morris's experiments prove that the highly parallelizable operation of matrix multiplication may offer speedups of 200% of the, compared to their software implementation. However, it is important to note that this statistic is dependent on factors such as system architecture and the algorithm itself.

FPGAs' utility lies within their ability to take some of the load off of CPU/GPUs, and perform multithreaded applications with minimal overhead. Much in the same way that transferring data to the GPU for computation incurs some overhead, transferring function calls to the FPGA requires a performance slowdown. Recent research involving the interplay between the FPGA and the CPU include papers on designing a way to identify and offload operations that potentially offer the most speedup when implemented in hardware from the CPU (which generally is accessed via software) to the FPGA (which requires knowledge of hardware design).

In their paper, "Dynamic Hardware/Software Partitioning: A First Approach"[3] , Greg Stitt, Roman Lysecky, Frank Vahid profile existing tools to identify most frequently executed application software. Stitt, et al., explain their method of dynamically partitioning a future software implementation in a SW/HW implementation. Yet we have yet to learn of a tool that can analyze if exporting these loops would be truly beneficial. Therefore, compared to pervious work which analyzed a programs usage frequency or properties such as "the three p's", our program analyzes runtime of software and hardware implementations and data transfer rate to determine if a program should be implemented in hardware.

In "A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning," [3] Roman Lysecky and

Frank Vahid found speedups ranging from 200% to 1000% by implementing a once purely software implementation into a hardware/software implementation of the program. By alleviating the CPU of a portion of the computation, speedups were found and they even discovered a reduction of power usage by 99%. This study is particularly interesting to our project, because we believe our testing program will be able to identify which functions of the program should be offloaded, and reduce some of the manual analysis with identifying which areas are best suited to offloading. This results in the overall system performance improvement and lower power consumption compared to a software implementation alone. While the speedup resultant is dependent on many characteristics of the program (such as the three p's), our research has shown the FPGAs have the potential to offer great speed up.

4 Our Idea

Our idea was to create an application that would take in as input a specific image processing algorithm implementation in hardware and software, fit the runtimes to a polynomial of a given degree, and dynamically determine whether to run it on the CPU's software or the FPGA's hardware.

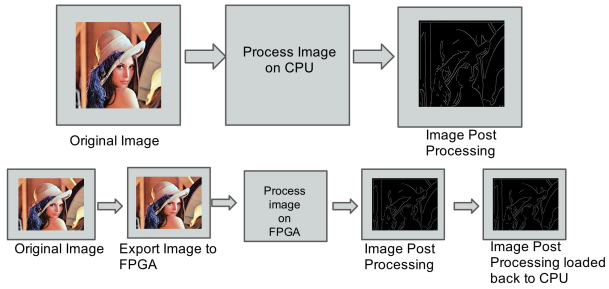

Figure 2: High Level Apporach to Exporting to FPGA

4.1 Relevance

Our method for determining the offloading point lies between the two bounds of using our test results of calculating the runtime for a set of image sizes and empirically searching for the point and using a model based to fit the data approach.

Empirical Method - the user has to provide a large set of increasingly bigger image sizes, a hardware implementation, and a software implementation. Using the three the application can iteratively run the set of images on both the CPU and FPGA. The application runs until the data points overlap, once this occurs, the offloading point has been found. The user can then determine if indeed the FPGA runs faster for their specific use case.

Model Based Method - the user has to provide parameters that describe the environment and the task being implemented (such as the rate of transfer to the FPGA) the image processing implementation algorithm runtime, the size of the input image, *etc* ... These parameters are inputs to the model, which then is fitted based off of these variable and returns optimal decision based on the fit and the values given by the user.

Our Method - the user needs a smaller set of increasingly bigger image sizes, the degree of the polynomial fit for the run times of the hardware and software implementations, and a hardware and a software implementation. The user only has to run the tests on the CPU to find the software runtimes. These runtimes and the bit rate of transfer to the given FPGA are used to create a model, which will determine the runtime for any given image. Using this the user can then determine the offloading point by extrapolating the data and seeing where the CPU runtime is equal to the Hardware time.

4.2 Implementation

To determine the offloading point we set the software time equal to the hardware time. To generate a software time relation, we timed our software implementation on 5 sets of three test images, and used the equation for the quadratic curve that fitted the data best. For hardware time, we used the lower end of Roman Lysecky and Frank Vahid's hardware speedup estimation, and said it is equal to half the software time (the computation done on the CPU) plus the overhead associated with offloading and onloading the data to the FPGA. This converts to the following equation, which can then be solved for x. X's value is the size at the offloading point:

$$ax^2 + bx + c_1 = \frac{1}{2}(ax^2 + bx + c_1) + (c_2 + c_3)x$$
$$2ax^2 + 2bx + 2c_1 = ax^2 + bx + c_1 + 2x(c_2 + c_3)$$
$$ax^2 + (bx - 2x(c_2 + c_3))x + c_1 = 0$$
$$ax^2 + [b - 2(c_2 + c_3)]x + c_1 = 0$$
$$\frac{-[b-2(c_2+c_3])\pm\sqrt{[b-2(c_2+c_3)]^2-4ac_1}}{2a}(eq1)$$

4.3 Hardware

Our FPGA board is an Altera DE2 Cyclone II. We programed the board via the Quartus II environment to generate hardware configurations using Verilog. Successfully implementing a hardware version of our image processing algorithm would have enabled us to get hardware runtime results as done with software. These data points could have been used to find a fit for the runtime of the program, as opposed to an estimate speedup which we used. Due to the implementation problems that we had, our team simply performed data transfer to and from the board, and timed these transfers. Our implementation read and wrote data to and from the SRAM memory of the DE2 board, and used the USB to transport data, see fig. 1.

4.4 Software

In our software implementation, we created an application that ran our algorithm, recorded the times, fitted them to a curve, then calculated the offloading point for that algorithm. The first part of the application was a python file that performed an $O(n^2)$ image processing algorithm. It read a file from memory, and then based on the user's preferences, either inverted the colors, applied a red filter, or converted the image to black and white. The application also timed the program runtime, minus reading and writing the image to disk. We were then able to write another python program that took that time data as an input, and then find the equation for curve of best fit. Finally, we used the coefficients from that curve and substituted them into equation 1 along with the FPGA data transfer bitrate, and solve the equation for the offloading point.

4.5 Results

The image processing algorithm was run on the CPU with our test set of images. We considered the runtime of the algorithm to be the time it took to read every pixel from the input image and change its value. For each of our two implementations and for each of our three images size we recorded 5 runtimes. The three images sizes were 38KB, 300KB and 900KB in a bmt format.

4.6 CPU

After running the software implementation five times on the CPU and recording a runtime each time we plotted the fifteen points using MATLAB. As shown in Figure 3 the points were fitted to a quadratic curve.

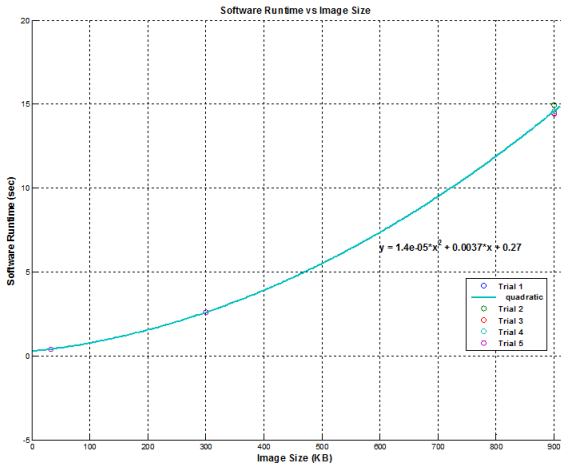

Figure 3: Software Runtime vs. Image Size

This curve describes the relationship betwen the software runtime and the size of the input image size. From this curve we are able to extrapolate and determine the runtime for any given image size.

4.7 FPGA

To determine the overhead associated with offloading and onloading from the FPGA we measured the time it took Quartus to write to and read from the FPGA. For both offloading and onloading, we measured five runs for all three images size for a total of fifteen points each. In both cases the points were plotted and the lines of best fit were found. These lines were used to predict the data transfer time for any size image.

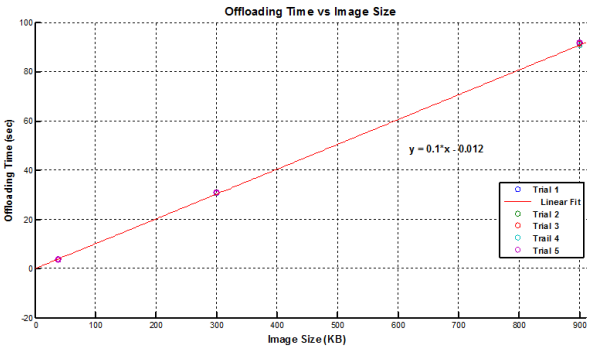

Figure 4: Offloading Runtime vs. Image Size

Figure 4 shows the relationship between the image size and the runtime. From this graph it is obvious that there are three distinct clusters that determine

the linear relationship between the input size and the runtime of offloading.

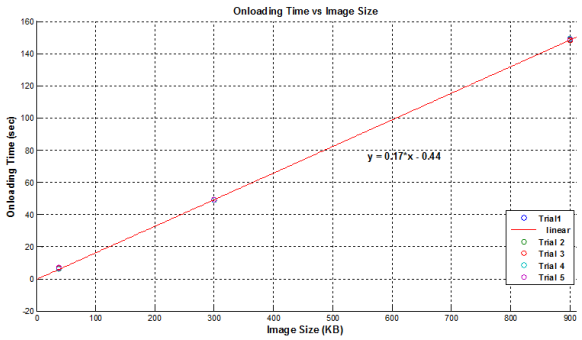

Figure 5: Onloading Runtime vs. Image Size

Figure 5 shows the same three clusters that correspond to the three different image sizes. These clusters again determine the linear relationship between the runtime of onloading and the input image size.

5 Evaluation

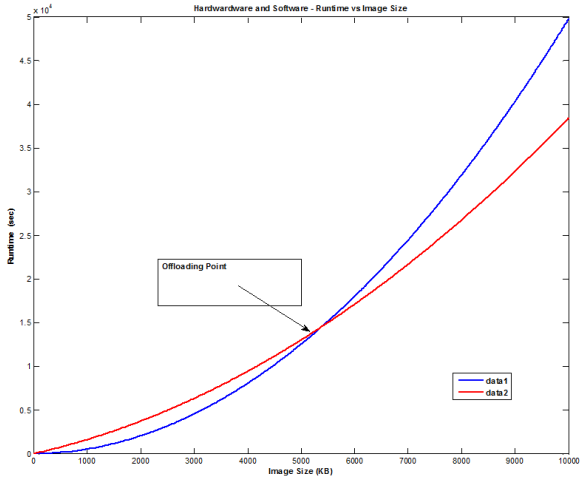

Figure 6: Software and Hardware Runtime vs. Image Size

6 Conclusions and Future Work

Our team developed a program which takes an image processing program as input, and from this runs the program on 3 different sized images. Currently the user can only decide upon one polynomial degree (our testing assumes quadratic fit, but this can easily be changed). Additionally we only know the transfer rate specific to the board we tested with. The speedup achieved by the hardware was simply an estimation.

In any future work, we would like to test the fit of the data to many polynomial degrees and have the program dynamically choose the best fit and from this calculate the offloading point. Additionally, getting test data from the hardware implementation would greatly increase the accuracy of our offloading point estimation. Getting all of the aforementioned aspects functioning, our improved program could take an input hardware and software implementation, fit the runtime data gathered to the curve that most closely fits the data, and from this calculates the offloading point. We would also like to extend the test set to a wider range of input sizes for empirical verification.

7 Acknowledgements

Our group would like to thank Benjamin Ylvisaker for his guidance, patience, and advice through the process of carrying out our project.

References

[1] Introduction to fpga technology: Top 5 benefits, 2012.

[2] M.D. Edwards and J. Forrest. Hardware/software partitioning for performance enhancement. In *Partitioning in Hardware-Software Codesigns, IEE Colloquium on*, pages 2/1–2/5, 1995.

[3] R. Lysecky and F. Vahid. A study of the speedups and competitiveness of fpga soft processor cores using dynamic hardware/software partitioning. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 18–23 Vol. 1, 2005.

[4] G.R. Morris. Floating-point computations on reconfigurable computers. In *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, pages 339–344, 2007.

[5] A. Shrivastava, S. Kumar, H. Kapoor, S. Kumar, and M. Balakrishnan. hardware/software partitioning for concurrent specification using dynamic programming. In *VLSI Design, 2000. Thirteenth International Conference on*, pages 110–113, 2000.

[6] Gupta. Sparsh, Mittal. Saket and Dasgupt S. Fpga: An efficient and promising platform for real-time image processing applications, 2008.

[7] G. Stitt, R. Lysecky, and F. Vahid. Dynamic hardware/software partitioning: a first approach. In *Design Automation Conference, 2003. Proceedings*, pages 250–255, 2003.

Integrating KLEE and Daikon for Enhanced Software Testing

Aidan Shackleton and Sophie Libkind

Department of Computer Science, Swarthmore College
ashackl1@swarthmore.edu
slibkin1@swarthmore.edu

Abstract. The advent of dynamic software analysis has provided new methods and insight into the automatic generation of test suites and the detection of program invariants. We propose that these two tasks can be integrated to increase the efficiency and breadth of both. Using the KLEE symbolic virtual machine and the Daikon invariant detector, we analyze the feasibility of probabilistic invariant detection through automatically generated tests and invariant-backed test generation. We use symbolic execution to evaluate the robustness of invariant detection with respect to incomplete testing suites. We also use knowledge of program invariants to streamline the process of symbolic execution. Ultimately we find that even with limited testing suites, Daikon approximates program invariants well. We also describe possible modifications to KLEE's symbolic execution process to make effective use of program invariants. Lastly, we propose a method of integrating KLEE and Daikon for automatic test generation.

1 Introduction

The execution of a program can follow different paths depending on the program's inputs. Two related challenges faced by software engineers are:

1. Does the program behave correctly on all inputs? That is, does every possible execution path do what it is intended to do?

2. What statements can be made about the program as a whole, regardless of execution path? That is, what program invariants exist over all inputs?

The approaches used to solve these challenges have historically been quite distinct, the former relying on exhaustive application of manually-generated test suites and the latter on careful, proof-based analysis of program structure. In recent times, tools falling under the umbrella of dynamic analysis have lead to the development of automated techniques for testing, test suite generation, and invariant detection. Interestingly, these techniques are closely related. Symbolic program execution partitions the space of possible program inputs based on the path of execution taken by the program on those inputs, reducing the set of tests necessary for complete program coverage to a single test for each cell of the partition. Each cell represents a deterministic execution of the program conditioned on a set of invariants (i.e. the restrictions on input required for membership in that cell). Dynamic invariant detection, meanwhile, traces the execution of the target program on a set of inputs (such as a test suite), using an inference generator to determine what conditions hold on all executions. We make two observations regarding the interaction of symbolic execution and invariant detection:

1. Knowledge of program invariants can inform the process of symbolic execution. Specifically, limiting the scope of variables may drastically reduce the number of distinct inputs that need to be considered, speeding up the (generally expensive) execution.

2. Invariant detection is imprecise without an exhaustive test suite, but even with automated methods, such a test suite can be prohibitively expensive to create. Incomplete test suites may still result in useful invariants, but manually-generated suites are likely to have systematic gaps in coverage. Symbolic execution creates the potential for randomized incomplete test suites, which may allow for probabilistic detection of program invariants.

As these observations suggest, there is significant potential for integrating existing systems for symbolic execution and invariant detection. Such integration can improve the efficiency of tasks that are currently cost-prohibitive, while also creating opportunities for new techniques for a variety of tasks.

In this paper, we make use of the KLEE symbolic virtual machine [2] and the Daikon dynamic invariant detector [4] to carry out two tasks motivated by the above observations. In particular, we use invariants generated by Daikon to annotate target programs with assert statements and observe the perfor-

mance of KLEE run on these augmented programs. Further, we use the complete test suite generated by KLEE to infer the robustness of Daikon with respect to the completeness of test suites and to determine the viability of detecting program invariants with a randomized partial test suite. In light of these experiments, we propose a method of integrating KLEE and Daikon with the goal of more complete and efficient testing.

In Section 2 we describe the KLEE and Daikon systems in more detail. In Section 3 we outline our methods for integrating the KLEE and Daikon systems. We present the results of our tests and discuss their implications in Section 4. Section 5 gives an overview of related work, and Section 6 concludes.

2 Background

In this section we discuss the two systems, KLEE and Daikon, which dynamically implement symbolic execution and invariant detection, respectively. In particular, we focus on the aspects of the KLEE and Daikon that complement each other.

2.1 KLEE Symbolic Virtual Machine

KLEE [2] is a dynamically implemented symbolic execution tool that aims to (1) cover every executable line of code and (2) detect program traces which will execute "dangerous" operations such as `assert` statements. Furthermore, KLEE is capable of providing a concrete test input that leads to each potential error.

KLEE runs a program symbolically, meaning that each operation in a particular program trace adds to a set of constraints that represents all possible input values leading to that trace. Once the program trace reaches an error or exit operation, KLEE solves the set of constraints to produce a concrete input value that will deterministically follow the program trace or reports that no such value exists. KLEE explores every program trace until either all branches have been explored or a user-specified time is exceeded.

For example, consider the subroutine `bar(x,y,z)` described in Figure 1 and the program trace of `bar` that leads to `foo2`. At each branch of the program, KLEE tracks the input constraints which correspond to this trace. In order of observation, these constraints are `x < 0`, `y < 0`, and `!(z * z < 0)`. At exit, KLEE's constraint solver will produce a concrete input such as `x = -1`, `y = -1`, `z = 1` that satisfies these constraints. However, observe that such an input may not always exist. For example, the program trace leading to `foo1` generates the constraints `x < 0`, `y < 0`, and `z * z < 0`, which is unsolvable because an integer squared is always greater than or equal to 0.

```
bar(x,y,z):
  z = z * z
  if x < 0:
    if y < 0:
      if z < 0:
        foo1(x,y,z)
      else:
        foo2(x,y,z):
    else:
      if z < 0:
        foo3(x,y,z)
      else:
        foo4(x,y,z)
  else:
    if y < 0:
      if z < 0:
        foo5(x,y,z)
      else:
        foo6(x,y,z):
    else:
      if z < 0:
        foo7(x,y,z)
      else:
        foo8(x,y,z)
```

Fig. 1: A program that takes integer inputs `x`, `y`, and `z` and has 8 distinct execution paths.

In the evaluation of KLEE on the GNU Coreutils and Busybox suites, KLEE had high (average 90%) code coverage [2] and revealed errors that had eluded developer tests for 15 years. These experiments also revealed a high overhead: each utility was run for 60 minutes and many did not finish in that time frame.

KLEE's primary strength in the context of testing is that it builds a concrete test suite with high code coverage. High code coverage suggests that the test suite also has broad coverage of the execution tree; unlike manually-generated tests, which are limited by the imagination of the programmer, KLEE has no bias in the execution branches it tests.

Unfortunately, KLEE is weak in other respects. Exploring all execution paths is computationally expensive and possibly inefficient depending on the program implementation. For example, to achieve complete coverage of `bar(x,y,z)`, KLEE must explore all eight branches of the execution tree (Figure 2) even though only four of the branches are reachable and lead to concrete test inputs.

Since `!(z * z < 0)` is an invariant of this program, augmenting the execution path by check-

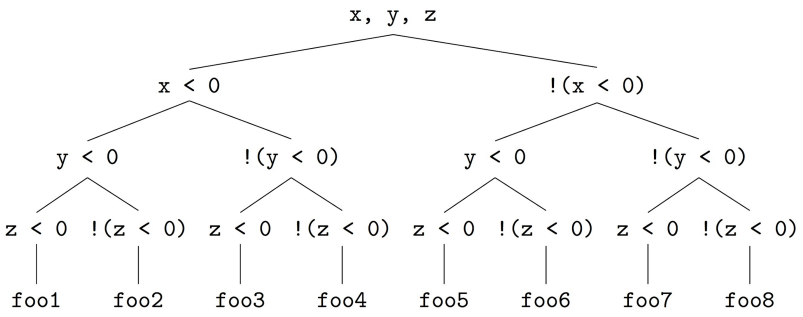

Fig. 2: The execution tree for the program `bar(x,y,z)` shown in Figure 1.

ing for the possible values of `z * z` immediately prior to the `if` statements (for example, by adding an `assert(!(z < 0))` statement at line 2 of `bar`) changes the execution tree to the one depicted in Figure 3.

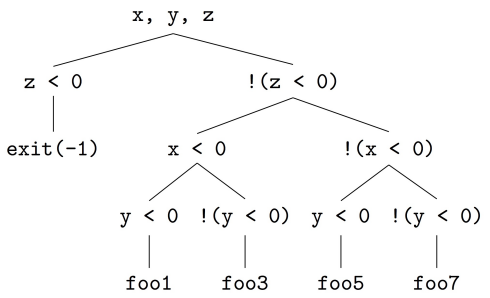

Fig. 3: The execution of `bar` modified by asserting the sign of `z` prior to branching.

With this augmentation, KLEE explores a significantly smaller tree and still returns the four test inputs corresponding to reachable branches of the program execution.

2.2 Daikon Invariant Detection

Daikon [4] dynamically detects likely program invariants, properties that hold true for every execution of a program. These invariants are expressed by a broad grammar of properties and variables. By default Daikon checks 75 different properties, such as is constant ($\texttt{x} = \texttt{c}$), in a range ($\texttt{a} \leq \texttt{x} \leq \texttt{b}$), or sortedness (`ls.isSorted()`), and allows the user to expand the list of properties. Daikon allows properties over a wide variety of variables including function variables, function paramenters, return values, global variables, and results of observer methods. As part of its dynamic invariant detection process, Daikon also implements a system for detecting conditional invariants. A *conditional invariant* is an invariant that depends on some other value or property: "If $x = 0$, then function returns an empty list" and "After n loop iterations, the first n elements of the list are sorted" are both conditional invariants.

Daikon infers invariants including conditional invariants through the inspection of multiple traces of a program execution. Properties that hold for each trace are reported to be invariants that hold in general. Daikon uses dynamic instrumentation to track variable values, feeding the result into an inference generator that creates a variable grammar and applies a machine learning process to derive a set of likely invariants. Accurate reporting requires that the tested program traces (corresponding with different test inputs) have high coverage of the program branches in its subroutines.

Daikon has potential uses for documentation, avoiding bugs, debugging, formal verification, and improving test suites. The last use case is the primary focus of our application of Daikon.

```
foo(x,y):
  y = y * y
  if (x < 0):
    if (y < 0):
      z = -1
    else:
      z = 0
  else:
    z = 1
  return z
```

Fig. 4: A program that takes integer inputs `x` and `y`.

Unfortunately, Daikon is limited by its dependence on the test suite used to for invariant detection. We illustrate this problem more fully with an example.

Consider the program `foo(x,y)` in Figure 4. Given this program and the test suite:

- `x = -1, y = 1`
- `x = 1, y = 0`
- `x = 2, y = -1`

Daikon will infer that the following invariance and conditional invariance hold at the program's exit:

- `!(y < 0)`
- `z = 0 or 1`
- `x < 0` $\Rightarrow$ `z = 0`
- `!(x < 0)` $\Rightarrow$ `z = 1`

In this case, these are indeed program invariants and will hold on every input. The accuracy of these invariants is a consequence of the completeness of the test suite; the three tests explored every reachable line of code in `foo`. On the other hand, consider the following test suite:

- `x = -1, y = 1`
- `x = -1, y = 0`
- `x = -2, y = -1`

This test suite is incomplete since none of the inputs reach the `else` case in which `z = 1`. Using these tests, Daikon will infer that the following invariants hold at the program's exit:

- `x < 0`
- `!(y < 0)`
- `z = 0`

In fact, only the second of these invariants holds for all possible inputs. The other two fail on any input in which `x` is non-negative. Motivated by examples such as this, we are interested in evaluating the ways in which and the magnitude to which Daikon errs when given only a partial test suite. Such an evaluation is particularly interesting as developer test suites may systematically avoid branches of the execution tree [2], while complete suites may be computationally infeasible to generate. If Daikon performs poorly under partial test suites, then it is severely limited by a program's complexity.

3 Methods

In this section we outline two experiments examining the possibilities for integrating KLEE and Daikon. The first attempts to use the output of Daikon to speed up KLEE's execution. The second analyzes the quality of invariants produced by running Daikon on a random subset of KLEE's output.

3.1 Using Daikon to Inform KLEE

As mentioned in Section 2.1, we believe that knowledge of program invariants can refine and expedite symbolic execution. This knowledge is particularly important when branch conditions are difficult for the symbolic machine to analyze. While difficulty of analysis is unavoidable (in general, determining the satisfiability of a conditional statement is NP-hard), invariants can simplify statements to the point that they are manageable. Moreover, the limitations of KLEE's constraint solver are such that knowledge of invariants may allow KLEE clearer navigation in branches that previously involved unsolvable constraints.

We run a simple experiment to evaluate the extent to which making invariants explicit speeds up KLEE's execution on branch-heavy functions. A visual outline of this experiment is given in Figure 5. We develop functions, similar to the one presented in Figure 1, that are composed of nested `if` statements such that many of the branches are unreachable and time KLEE as it symbolically executes them. For each program, we then run the resulting test suites through Daikon, developing a set of invariants. After making invariants explicit by adding `assert` statements to the program, we rerun KLEE on the modified programs and compare both the execution time and number of tests generated to the original run.

This experiment accomplishes two goals. First, if adding `assert` statements decreases the execution time of KLEE without reducing the number of tests generated and without violating the `assert` statements, it shows that forcing KLEE to explicitly acknowledging invariants can benefit to test suite generation. Second, by examining programs of different sizes without changing the difficulty of solving the constraint set, we can determine how much time KLEE spends solving constraints versus traversing the execution tree.

3.2 Probabilistic Invariant Detection

The goal of probabilistic invariant detection is to evaluate the effectiveness of generating program invariants using Daikon on a partial test suite. As observed in Section 2.2, the completeness of the test suite that Daikon uses to infer invariants affects Daikon's output. In particular, an incomplete test suite (one that does not cover all branches of the execution tree) may cause Daikon to:

1. over-generalize application of invariants: infer invariants that hold for all inputs in the incomplete suite but not in general. And,
2. over-specify invariants: fail to generate invariants that hold in general because Daikon focuses on too-specific invariants that are implied by actual program invariants but are not general enough.

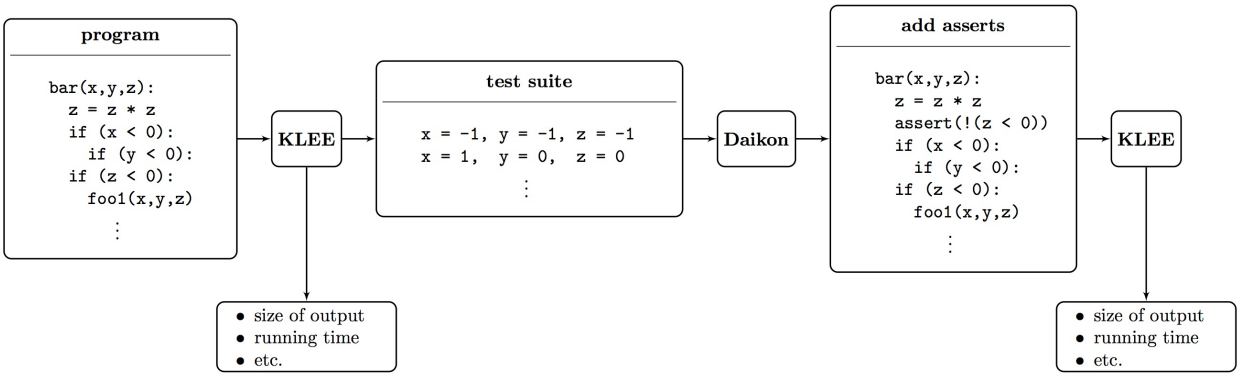

Fig. 5: A proposed method of comparing the performance of KLLE with and without knowledge of Daikon-generated invariants.

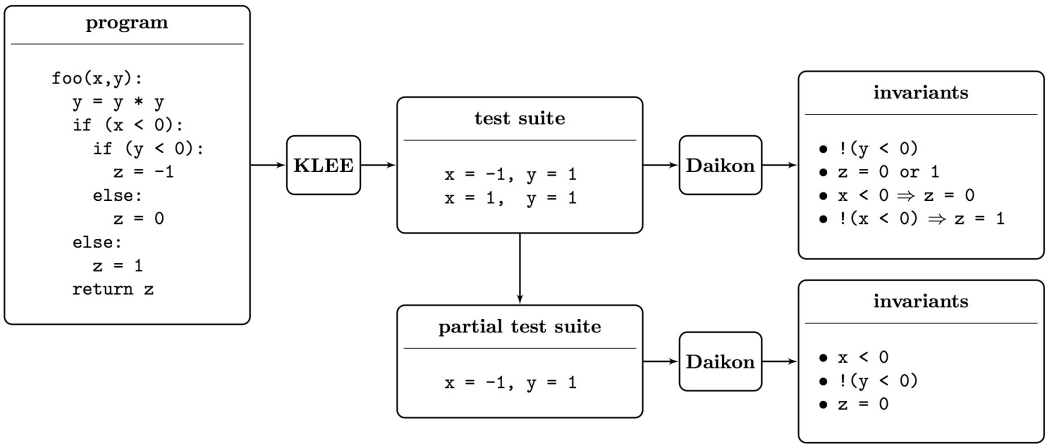

Fig. 6: The architecture of the experiment to evaluate Daikon's performance using an incomplete test suite. KLEE is used to generate a complete test suite. Daikon infers invariants based on these tests and separately on a subset of these tests. The results are compared.

We evaluate the degree to which Daikon is susceptible to these errors by comparing the invariants generated by Daikon using a KLEE-generated complete test suite and the invariants generated by Daikon using a subset of this complete suite (Figure 6).

We evaluate probabilistic invariant detection with respect to three different types of programs: sorting algorithms (selection sort, insertion sort, merge sort, and quick sort), arithmetically complex algorithms (computing the relative orders of a and b in $\mathbb{Z}_p$, computing the greatest common denominator of a and b using the Euclidean algorithm, computing the greatest common denominator of a and b using prime factorization, and computing the discrete logarithm $\log_a b$ in $\mathbb{Z}_p$ using the Tonelli-Shanks algorithm), and recursive algorithms (factorial, computing the sum of the absolute value of list elements, computing the number of distinct primes in the prime factorization of n, and computing if a list is a palindrome). These programs vary in complexity when measured with respect to program invariants and size of execution tree. The number of invariants detected using a complete test suite vary from 15 (recursive factorial of $n < 100$) to 712 (discrete logarithm in $\mathbb{Z}_{11}$ and $\mathbb{Z}_{13}$ using Shanks algorithm) and the number of KLEE-generated tests (which are in 1-1 correspondence with reachable execution paths) vary from 24 (recursive palindrome on lists with size between 6 and 10) to 1024 (recursive sum of the absolute value of list elements on lists of size 10).

4 Results

This section presents the results of our two experiments.

4.1 Results for Using Daikon to Inform KLEE

We ran KLEE five times on a function with 16 reachable branches out of 32 total branches, and five times on a function with 512 reachable branches out of 1024 total branches. In both cases, the unreachable branches were caused by checking the condition `z < 0` after squaring the variable `z`. We then added the statement `assert(!(z < 0))` prior to branching and repeated both experiments.

In both cases, the addition of the assert statements made no significant difference in the runtime of KLEE. In the 32-branch case, KLEE averaged 21.1238 seconds before adding the `assert` statement and 21.1344 seconds afterwards. In the 512-branch case, KLEE averaged 23.9218 seconds before adding the `assert` statement and 23.9210 seconds afterwards.

These results demonstrate two important properties of KLEE. First, changing the order in which the program execution encounters conditional branches has no perceptible impact on KLEE's execution time. It is likely that in the pre-`assert` case, KLEE stores the constraint `!(z * z < 0)` either upon executing the statement that squares `z` or upon reaching the `z < 0` condition for the first time. Thus, prompting KLEE to consider known invariants early in its execution does little to change its runtime. Second, KLEE spends the vast majority of its execution time on these programs evaluating the fairly basic rule `!(z * z < 0)`, so much so that computing an extra 496 test cases is fairly negligible in comparison. This result implies that knowledge of invariants could still be useful in reducing execution time, though incorporating such knowledge is more complicated than adding a series of `assert` statements to each function.

In future work, we propose a modification of KLEE to allow a special statement to add constraints to the set KLEE maintains as it executes the program. The difference, with respect to `assert` statements, is that KLEE would immediately accept these constraints as true rather than attempting to find examples invalidating them. While such statements would not be risk-free—adding invariants that do not actually hold over the entirety of the execution tree would prevent KLEE from executing correctly—they allow KLEE to bypass complex computation on statements that are verifiable via known invariants. This modification has the additional benefit of allowing programmers to restrict KLEE's symbolically-generated inputs to a particular range within the program.

4.2 Results for Probabilistic Invariant Detection

For each of the twelve programs described in Section 3.2, we run Daikon on randomly selected subsets of the complete KLEE-generated test suite. We choose subsets with sizes in increments of 10% of the size of the complete suite. In each case we ensure a finite KLEE-generated test suite by limiting the range of any integer arguments and the size of any list arguments. We compare the resulting invariants to the invariants generated by Daikon using the complete test suite and determine the average recall (Table 1) and precision (Table 2) over 20 runs.

Low recall corresponds to Daikon's tendency to over-generalize, while low precision corresponds to Daikon's tendency to over-specify. We anticipate that

program	total invariants	execution paths	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
selection sort	125	33	0.3828	0.3956	0.7768	0.8664	0.8696	0.9196	0.9356	0.9616	0.9796
insertion sort	167	120	0.6970	0.7533	0.7856	0.8222	0.8341	0.8886	0.9012	0.9449	0.9689
merge sort	190	120	0.9060	0.9222	0.9364	0.9404	0.9637	0.9575	0.9775	0.9858	0.9968
quick sort	201	120	0.9264	0.9473	0.9527	0.9577	0.9701	0.9806	0.9831	0.9786	0.9915
relative orders	63	104	0.5492	0.6286	0.6952	0.8063	0.8571	0.8762	0.9238	0.8968	0.9413
gcd by Euclidean alg.	76	360	0.7276	0.8375	0.8961	0.8217	0.9112	0.9204	0.9592	0.9039	0.9217
gcd by factorization	170	159	0.7212	0.7406	0.7982	0.7897	0.8594	0.8279	0.8544	0.8744	0.8729
discrete logarithm	712	235	0.7057	0.7582	0.7668	0.8076	0.8277	0.8827	0.8864	0.8611	0.9029
factorial	15	101	0.1853	0.2707	0.3413	0.3933	0.6880	0.7053	0.6880	0.7227	0.9293
sum of list elements	34	1024	0.9941	0.9941	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
distinct primes	64	49	0.6062	0.8625	0.8797	0.9266	0.9016	0.9453	0.9719	0.9672	0.9750
palindrome	78	24	0.4051	0.7295	0.9462	0.9769	0.9833	0.9821	0.9885	0.9962	0.9949

Table 1: Recall of invariant detection using different proportions of the KLEE-generated test suite in increments of 0.1. Average over 20 runs.

program	total invariants	execution paths	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
selection sort	125	33	0.6632	0.8504	0.8324	0.8844	0.8913	0.9293	0.9451	0.9705	0.9863
insertion sort	167	120	0.5960	0.6830	0.7375	0.7727	0.8043	0.8480	0.8694	0.9228	0.9518
merge sort	190	120	0.7613	0.8660	0.9060	0.9186	0.9586	0.9654	0.9802	0.9892	0.9950
quick sort	201	120	0.7291	0.7762	0.8135	0.8567	0.9108	0.9355	0.9343	0.9628	0.9765
relative orders	63	104	0.7972	0.8082	0.8622	0.8625	0.8926	0.8976	0.9297	0.9128	0.9324
gcd Euclidean alg.	76	360	0.8547	0.9458	0.9722	0.9615	0.9795	0.9838	0.9851	0.9906	0.9979
gcd factorization	170	159	0.7990	0.8600	0.9032	0.9025	0.9200	0.9330	0.9389	0.9694	0.9763
discrete logarithm	712	235	0.6011	0.6576	0.6583	0.6886	0.7241	0.7339	0.7378	0.7325	0.7611
factorial	15	101	0.4964	0.6152	0.6919	0.7375	0.9053	0.9121	0.9053	0.9186	0.9817
sum of list elements	34	1024	0.8244	0.9185	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
distinct primes	64	49	0.6361	0.7263	0.7537	0.8317	0.8863	0.8631	0.8736	0.9036	0.8703
palindrome	78	24	0.5430	0.6982	0.7944	0.8355	0.8877	0.9108	0.9542	0.9676	0.9773

Table 2: Precision of invariant detection using different proportions of the KLEE-generated test suite in increments of 0.1. Average over 20 runs.

low recall is more detrimental to Daikon's usefulness, since it is easier for human programmers to weed out false positives than to determine new program invariants [3].

Our results show that both precision and recall appear to increase at a sublinear rate with respect to the proportion of test inputs. In other words, as more test cases are added, the marginal benefit of adding additional test cases decreases. This phenomenon is exemplified by the recall and precision of the three test programs (one from each category) shown in Figure 7and Figure 8. Furthermore, recall and precision are both reasonably high even when using only a small proportion of the complete test suite to detect invariants. Although there is no formal definition for a good approximation of invariants, in all but three of the programs (greatest common denominator by prime factorization, discrete logarithm, and factorial) over 90% of invariants are detected using only 70% of the complete test suite. Similarly, in all but three of the programs (insertion sort, number of distinct primes, and discrete logarithm), fewer than 10% of the invariants that are detected using only 60% of inputs do not hold for the complete test suite.

The performance of probabilistic invariant detection on the program that computes the sum of the absolute value of list elements provides further insight into the relationship between invariants and symbolic execution. Computing the sum of list elements has very few program invariants (34) relative to the number of execution paths (1024). This ratio suggests that each branch of the execution tree is similar in that the same invariants hold on each branch. If it were the case that the invariants were not consistent across the execution tree, Daikon would interpret these differences as conditional invariants. Therefore, it is not surprising that Daikon correctly infers all of the program invariants for the sum of list elements using on 30% of the complete test suite. In future work, it would be interesting to test this hypothesis on other programs with low ratios of invariants to execution paths.

On the other hand, it is not clear from our results which program properties make it difficult for Daikon to well-approximate program invariants with a partial test suite. Although we hypothesize that the feasibility of probabilistic invariant detection is inversely related to the variability of different execution paths, it is unclear how this variability should be measured. Tables 1and 2do not suggest any correlation between the performance of probabilistic invariant detection and either the number of invariants or the number of execution paths. One possible direction for this research is to explore a measure of invariant variability in a program's execution tree and how this measure relates to Daikon's ability to approximate invariants with an incomplete test suite.

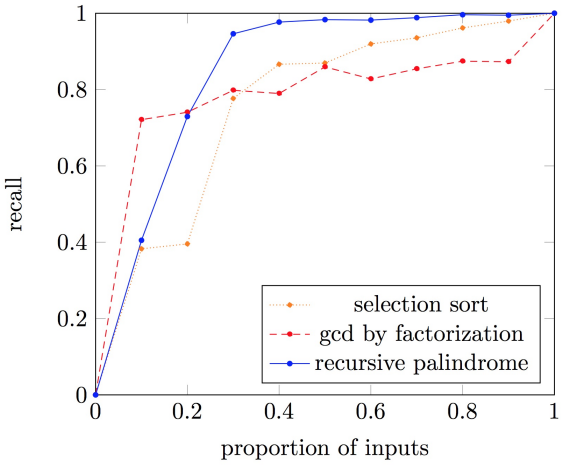

Fig. 7: Recall of invariant detection for programs of each type: selection sort of lists of size 4, computing the greatest common denominator of $a < 15$ and $b < 15$ using prime factorization, computing if a list with size between 6 and 10 satisfies the palindrome property. Numbers displayed are an average over 20 repetitions.

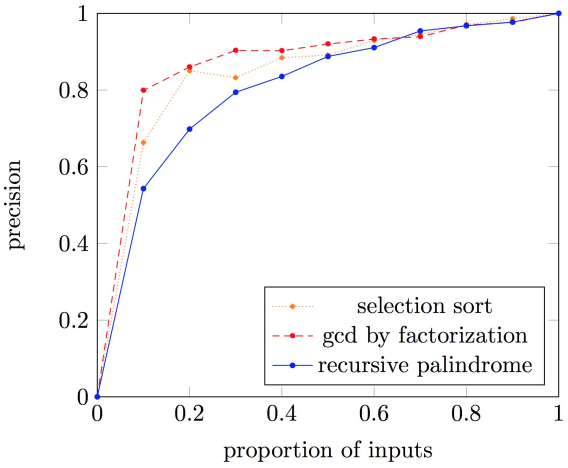

Fig. 8: Precision of invariant detection for programs of each type, averaged over 20 repetitions.

5 Related Work

Generating test suites. The Eclat tool [6] incorporates a small fault-revealing subset of test inputs from the larger set of tests, allowing the developer

to focus on a few useful inputs. As part of this technique, redundant tests are discarded. This procedure corresponds to choosing one input on each faulty execution branch. The larger set of tests may come from the developer test suite, or, alternatively, Eclat provides a technique for generating legal inputs.

Previous work on integrating static and dynamic analysis tools to generate test inputs (primarily [1]) have focused on using static analysis to guide dynamically-driven test generation toward program vulnerabilities. Although it uses different techniques (in particular, a visibly pushdown automata), this system incorporates similar ideas to ours while aiming to produce test cases exposing program vulnerabilities as opposed to a generic high-coverage suite of tests.

Substra [7] guides test generation using program constraints inferred by Daikon and based on subsystem states and define-use relationships. These constraints, as well as the sequence in which they occur, define valid input sequences that may appear in real-world applications.

Improving symbolic execution. An evaluation of various symbolic execution techniques [2] suggests that scalability is a significant limitation of symbolic execution techniques for test generation. Various optimizations to KLEE and other symbolic execution tools such as KLOVER [5] provide some improvement.

Evaluating invariance. Previous evaluations of Daikon's invariant detection system has primarily focused on the evaluation of conditional invariants and, in particular, on the relationship between splitter selection and generated invariants [3]. Although this evaluation compares different methods of splitter selection, it does not examine how these methods are affected by the test cases used to infer invariance.

6 Conclusion

Motivated by the complementary advantages and disadvantages of symbolic execution and invariant detection in the context of dynamic analysis, our research aimed to integrate these two tools in order to make the automatic generation of complete test suites more efficient. Using the KLEE symbolic virtual machine for symbolic execution and the Daikon invariant detection system, we determine that even though Daikon's output depends on the completeness of the test suite, it is generally robust with respect to partial test suites. In other words, it can be expected that a set of inputs not covering every branch of a program's execution tree is sufficient to guide Daikon to infer a mostly accurate set of program invariants. Related to this result, we propose the following direction of research: first define a measure of program variability between execution paths. Then evaluate the correlation between this measure and the precision and recall of probabilistic invariant detection.

Our second contribution explores the idea that knowledge of program invariants may guide the symbolic execution process. Although the results of this experiment are inconclusive, they suggest possible modifications to the symbolic execution process that use prior knowledge of invariants to decrease the number of branches KLEE must explore and consequently limit the computationally expensive process of constraint solving.

Finally, these two results suggest a method for augmenting an incomplete developer test suite. Since Daikon approximates invariants well in spite of test suite incompleteness, the program invariants produced by Daikon run on the developer tests will likely hold on all inputs. Furthermore, past research suggests that eliminating the few false positives would be a relatively trivial process for a human programmer. These invariants can be transformed into statements which can guide a modified version of KLEE through a simplified execution tree of the program, ultimately producing a complete test suite.

References

1. Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22. ACM, 2011.
2. Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
3. Nii Dodoo, Lee Lin, and Michael D Ernst. Selecting, refining, and evaluating predicates for program analysis. 2003.
4. Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.
5. Guodong Li, Indradeep Ghosh, and Sreeranga P Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In *Computer Aided Verification*, pages 609–615. Springer, 2011.
6. Carlos Pacheco and Michael D Ernst. *Eclat: Automatic generation and classification of test inputs*. Springer, 2005.
7. Hai Yuan and Tao Xie. Substra: A framework for automatic generation of integration tests. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 64–70. ACM, 2006.

Code Similarity: An Exploration in Plagiarism and Duplication Detection Systems

O'Connor, Melissa
moconno2

Pitser, Mallory
mpitser1

Yang, Jack
yyang4

December 20, 2013

Abstract

There are similarities that exist in code that is similar but not identical that humans can easily find but that computers sometimes have a hard time identifying. Tools that can identify which of many files contain similar code and which lines are similar would be helpful for many reasons, from amateur programmers trying to obtain better search results that accounts for the actual source code they have written, to experts searching for posted questions that contain code similar to code they have written so that they can help answer questions. Existing tools that perform these tasks are duplication detection tools used in industry such as CloneDR and Copy/Paste Detector (CPD) from PMD, which use techniques like string matching and comparing abstract syntax trees, and plagiarism detection tools used in academia such as MOSS and Sherlock which often use statistical techniques such as winnowing and tokenization.

We seek to identify the limits of some of the more commonly used tools, particularly in how accurately they can classify code as similar in ways that humans would see as relevant or related. We performed an experiment in which we compared a few plagiarism detection tools as well as a promising technique, code compression, that might be used to enhance results. We compared the ability of MOSS, Sherlock, and code compression with gzip to provide us with a ranking of code similarity that lined up with humans' rankings. We found that in most cases neither MOSS nor Sherlock was able to accurately discover which code snippets were similar. Code compression gave us promising results, but it is hard to imagine code compression as a stand-alone technique, given how barebones it is. Future work should be done to examine its usefulness given a wide variety of code and similarity types.

1 Introduction

The ability to identify similar sections of code is a difficult but important task. Detecting similar parts of code within or between projects has important applications in both academia and industry. There are two areas in which code similarity detection is particularly important, which involve finding exactly or close to exactly duplicated code and relatedly, finding instances of plagiarism. In industry, each line of duplicated code costs a company money to maintain, and bugs are more prone to be uncaught if there is duplicated code where a bugfix wasn't implemented. This has been the impetus to develop tools that can identify poorly factored code, which typically look for exactly duplicated code or code that has been minorly edited from some other version elsewhere in the project. In academia, concerns about plagiarism detection in source code has led to the development of tools that provide some estimate of how much code within two files may be plagiarized based on some similarity score generated by the tool. Given the sheer amount of submitted code that must be screened for plagiarism, there is a great motivation for some sort of automated system to assist in the manual work of reading through submitted source code. This motivation has led to a decent amount of work done to develop tools that can accuraately identify which pieces of code are potential plagiarism cases. Unfortunately, once tools were developed that did a 'good enough' job of ranking code as being potentially plagiarized, the development of new plagiarism detection tools has stagnated.

Besides these two use cases, we see a need for code similarity tools that are less specialized. In particular, it seems that finding code that relates to one's project would be helpful for a variety of reasons, from simply providing extra reference points of how someone else has written the same functionality, to narrowing down search results to include those with code that is most similar to that which a user has already written. These types of tools do not yet exist as stand-alone tools, though they may be minimally implemented and integrated into some search recommendation algorithms for websites which allow users to upload or host code (e.g. GitHub).

In this paper, we will review and summarize some popular techniques that are applied to detect code duplication as well as some algorithms that are used to detect plagiarism.

2 Our Idea

Originally we aimed to develop a system that allowed users to upload their own code so that it could be used in a search engine to find relevant code forum posts (specifically taken from StackOverflow). We felt a tool like this would be helpful in cases in which programmers are attempting to find relevant help posts, yet do not yet know the proper terminology. For instance, a new computer science student might be unsure how to word a question about what they are struggling with, so it might help them to be able to upload a segment of code directly and not worry about wording. Or, an experienced computer programmer might have an altruistic goal of helping out as many questioning programmers as possible and want a better way to find questions they could easily answer, given code they have previously written.

Through our examination of pre-existing tools, we found several that we deemed as useful to solve our task. MOSS, a plagiarism detection tool, in particular seemed like the solution to our code-matching task. We planned to combine the functionality of MOSS with the StackOverflow API, StackExchange, in order to build a working prototype.

However, we made an assumption regarding the ability of a specific plagiarism detection tool, MOSS, to cross into the realm of general code similarity identification. In practice, the tool we planned to use was unable to detect similarities in similar code if the code was not essentially a copy-and-pasted version of the original with only minor alterations such as identifier name changes and slightly different control flow while maintaining almost identical content and functionality.

After seeing a need for a more general tool to identify similar pieces of code that did not make any assumptions about the code being copied and then deliberately altered to mask the plagiarism, we decided to investigate code similarity detection techniques and the effectiveness and limitations of tools that implement them in order to come up with a better direction for future work in this topic to go in.

We have identified and examined a few existing tools to figure out which techniques work in various situations. There are many different tools that perform similarity detection in source code, including matching code exactly, matching code according to some predefined threshold, matching while ignoring variable names, and matching code in specific languages. In section III, we will discuss relevant background information. This includes existing tools and how accurately their results line up with what real humans think is similar. In section IV, we will present the experiemental setup. In section

V we will present the results and discussion. Finally, in section VI we will conclude with an overview of the general problems we encountered when trying to create systems based on the existing systems and discuss future directions guided by our results.

3 Background and Related Work

Despite our belief that it is important and useful to identify publicly-available code that is somehow 'similar' to an arbitary code sample, the fact that there are only a couple use cases where these tools are of necessity has led to the development of a few highly specialized tools that essentially work to detect similarity in the way that the specific use case needs it to work. Most of the tools that currently exist and are being used are great at detecting code that has been directly copied and has only minor alterations made to it. However, detecting similarities between different code fragments could mean either detecting that the two fragments are similar based on looking at the code itself, or it could mean identifying that two code fragments have similar or identical functions (e.g. FOR loops versus WHILE loops). For the purposes of this paper, we will discuss only code similarity detection tools that examine the code itself rather than the output or function.

We will now discuss the differences between the tools and techniques used to detect duplicate code in industry versus those used to identify plagiarized code in academia.

3.1 Code Duplication

Code duplication detection tools are common in industry due to the high cost of maintaining each line of code. In industry, these tools are used to identify code that is poorly factored and which might benefit from refactoring or procedure extraction [10]. Duplicate code typically arises from the 'copy-and-paste' style of programming in which a chunk of code is copied from one location and pasted in another, often with no or very minimal changes to the copied version of the code [10]. Identifying duplicate code must account for the fact that some (sometimes trivial) changes may have been made in one version of the duplicated code. The goal is to identify what are termed as 'code clones;' a code clone occurs when one code fragment is the clone of another as long as they are similar by some definition of similarity or duplication [10].

Roy et al. (2009) [12] suggested four types of similarities based on both textual and functional similarities can exist between any two code clones.

The first type refers to identical code fragments with variations only in whitespace, layout and comments. The second type contains syntactically identical code fragments that are only different in identifiers, literals, types, and variations included in the first type. The third type may also include some added or deleted statements. The final type occurs when two or more code fragments perform the same computation but are implemented differently. The difficulty to detecting code duplication increases from the first type to the last type, with most existing tools being capable of detecting the first two types of similaties fairly easily, and very few efficient tools existing that can fully handle the second two types.

There are many code duplication detection tools. Each tool has a specific focus on its potential use cases. But in general, we can classify the tools into four categories based on the information it can extract from the source code, and the analysis it performs. The four categories are textual, lexical, syntactic and semantic [12].

Textual approaches perform little or no pre-processing of the code submissions. On most instances, raw source code is used, and is divided into sub-strings. The result is a set of fingerprints. Then, the fingerprints are hashed, and by comparing hash values, the systems identify code fragments with the same hash values as clones.

Lexical approaches (also known as the token-based approaches) preprocess the source code by transforming it into a string of tokens, according to the lexical structure of the specific programming language in which the source code is written. This encoding abstracts away the concrete names and values of parameters. The tokens are then transformed into a suffix tree, and by analyzing the suffix tree, the systems can detect code clones. Dup used the lexical approach with a parameterized matching alorithm. It is most effective in detecting the first and second types of clones, and can sometimes detect the third type by a maneuver involving a sort of concatenation of the first and second types [12].

Syntactic approaches use a parser to convert source code into abstract syntax trees (ASTs), which can then be processed by either tree-matching or metrics-based algorithms.

Finally, semantic approaches usually use a program dependency graph to analyze the source code. The nodes of the graph represent expressions and statements, and the edges represent control and data dependencies [12]. This representation abstracts away the order in which statements occur, and is effective for the fourth type of clones.

3.2 Code Plagiarism

Code plagiarism may also arise from a copy-and-paste style of programming, though this time one assumes that the plagiarizer will deliberately change any elements of the code that do not alter its functionality. For example, a plagiarizer may change the whitespace and layout, rename identifiers, reorder code blocks and statements within code blocks, change control flow, etc [6].

In terms of how one detects that two files are similar enough that there is plagiarism, the types of similarity occuring can be categorized along the same lines as code duplication with the four types of code clones.

However, the tools are built with the assumption that a user of the tool(usually a professor) is not going to rely on the tool to identify all lines in the code that are plagiarized; rather, it is most important for the tool to simply identify files that are most similar to each other since a human component will have to be involved anyway.

Existing plagiarism detection tools can be categorized based on their algorithms. In this paper, we will evaluate and compare a couple source-code plagiarism detection tools that use different categories of algorithms. As suggested by Mozgovoy, plagiarism detection algorithms can be classified into three categories: fingerprint-based systems, string-matching systems, and parameterized matching algorithms [6].

Tools based on the fingerprint approach work by creating 'fingerprints' for each file which contain statistical information about the file. These fingerprints may include, but are not limited to, the following type of data: average number of terms per line, number of unique terms, and number of keywords. The systems create the 'fingerprints' by extracting and counting the attributes from source code submissions, and flagging the pairs of submissions which are suspiciously similar for humans to inspect further. However, the drawbacks to tokenization are that the tools created are inherently language-dependent, and tokenization can cometimes make two very different lines of code appear to be similar [6].

Many of the earliest plagiarism detection systems were fingerprint-based systems. In 1976, Ottenstein developed a tool for detecting identical and nearly identical student work [8], [7] using Halstead's software metrics to detect similarities by counting operators and operands for ANSI-FORTRAN modules [3], [4].

Extending on Ottenstein's work, Robinson and Soffa developed ITPAD (Instructional Tool for Program Advising), another plagiarism detection tool that incorporated new metrics to Halstead's metrics in order to improve

system performance [11]. In addition to counting the common attributes, ITPAD breaks each program into blocks and builds a graph representing the structure of each file submission. It then generates a list of attributes based on the lexical and structural analysis and compares pairs of submissions by counting these characteristics.

Some of the most well known and recent string-matching-based systems include MOSS [1], JPlag [9], Yet Another Plague (YAP3) [14], and Sherlock [5]. In most string-matching-based systems, including the ones mentioned above, there is an initial preprocessing stage called tokenization. At this stage, each substring in the source-code file is replaced with predefined and consistent tokens, such as identifiers, types, integers, strings, and so on; for example, different types of loops in the source-code may be replaced by the same token name regardless of their loop type (e.g. WHILE loop, FOR loop). This method is also used to replace variable names so that variation between files, based only on identifier names that are up to the user's discretion, is decreased. Each source-code document is then represented as a series of token strings. The tokens for each document are compared to determine similar source-code segments. Overall, this process effectively helps detect similar programs with consistent structures but renamed identifiers.

One of the earliest string-matching plagiarism detection tools, Measure of Software Similarity (MOSS), is one of the tools that we have studied more in depth. Instead of relying on a source file's lexical and semantic information, it is based on a statistical approach that divides programs into k-grams, where a k-gram is a contiguous substring of length k. Each k-gram is then hashed, and MOSS selects a subset of these hash values as the program's fingerprints, thus making MOSS a combination of string-matching and fingerprint-based systems. At a high level, source code similarity is determined by the amount of fingerprint duplication between the files – the more fingerprints they share, the more similar they are [13].

Finally, the DUP tool is based on a parameterized matching algorithm [2]. The algorithm detects identical and near-duplicate sections of the source-code by matching source-code sections whose identifiers have been substituted or renamed systematically.

4 Experimental Setup

Given that MOSS did not provide an accurate rating of which files were most similar in regards of our initial project concept, and after multiple failed attempts with other similar tools such as Sherlock, we decided to

perform a study of the bounds of these tools with respect to what a human would be able to identify as similar or related code.

Our goal through our rounds of experiments was to identify the strengths and weaknesses of several different code similarity tools. To test this, we identified two code samples taken from students in the introductory Computer Science course at Swarthmore College, that seemed typical of introductory programming assignments, and thus should have many related StackOverflow posts. One was recursively writing the Fibonacci sequence while the other was writing code for a simple game of hangman.

To start our experiment, we first identified a single highly relevant Stack Overflow post that contained code within the question portion. Next, we gathered more posts by using the original post's "Related Posts" section in the article. For the initial post and its related posts, we stripped out the code snippets from each to save in separate python files for testing. The search for Fibonacci sequence posts yielded 8 code samples; the hangman search contained 10.

To create a baseline for our comparisons, we ranked the posts manually against the base file to determine how "useful" they seemed and if there was a high degree of similarity. We defined "useful" to mean containing similar chunks of code and being marginally related to the same task, without much other irrelevant code.

Then, all files obtained from StackOverflow posts were run side-by-side against the base file (taken from the intoductory students' work) on two different code plagiarism tools, MOSS and Sherlock. We also performed a related experiment using code compression techniques.

The first plagiarism detection tool that we used to compare code was MOSS. This tool uses a string-matching algorithm called winnowing to return the number of line matches as well as the percentage of each file that these lines make up. Figure 1 is sample output from running the MOSS program.

The second tool was Sherlock which uses digital signatures to find similarities between files. A digital signature is a sequence of bits that have been joined into a number. It is formed by taking several words in an input file and turning them into a sequence of bits. Sherlock returns a percentage similarity index. We also ran Sherlock with a zero percent sensitivity threshold so it would detect any bit of similarity; otherwise, Sherlock only reports results when the similarity index is over twenty percent.

Lastly, we performed a code compression analysis in which we gzipped the individual base files as well as concatenated files that contained the base file code along with the StackOverflow post code. We then compared the

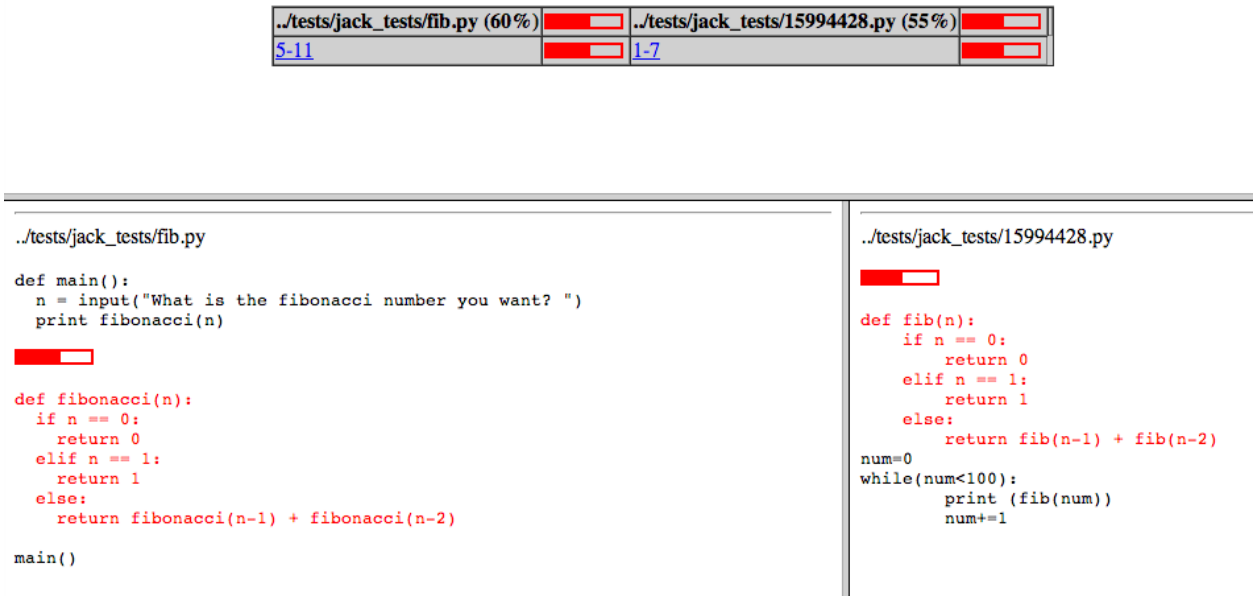

Figure 1: An example run of the MOSS Plagiarism Detection System.

concatenated file sizes to what would be expected. The expected value was determined as what the size would be if we added the compressed sizes of the two individual files composing the concatenated file. We then calculated the difference in these file sizes; presumably, the files with a greater difference should be similar because more code was able to be compressed.

5 Results and Evaluation

None of the three tools was able to accurately identify which code samples were similar in terms of matching up with how similar a human rates the code snippets. Most importantly, very few of the files that we ran actually were identified as having any similarity at all. Only two out of the eighteen file pairs we ran in MOSS and three out of the eighteen files we ran in Sherlock did not have a 0% matching rate. See Table 1 for a breakdown of the results, including the StackOverlow post IDs and the percentage scores for each file with each tool.

All of the files that contained matches were from the tests on the Fibonacci Sequence code, and all posts related to hangman had 0% matching rates.

Of the files that did contain matches when testing with Sherlock, two out of the three were highly ranked pre-testing, which meant we thought these files would help someone performing a search. For the MOSS results, one of the two files was highly ranked; the other would be deemed a false positive. This file contained a large amount of irrelevant code that a searcher would need to sift through to get their answer, leading to our low initial ranking. Unfortunately, there was also one other file that was highly ranked that showed no similarity and thus was a false negative. As far as we are concerned with our expectations for the code plagiarism tests, these results were unsatisfactory. We were hoping for some similarity to also be reported for the other files, which was not the case.

File ID	MOSS (lines matched)	Sherlock (%)
15305362	0	0
15515920	0	0
15611781	0	0
15820601	0	0
15994428	7	66
1678091	0	0
17624149	0	33
19871921	7	28
4935957	0	0
494594	0	0

Table 1: Fibonacci Sequence individual file sizes

In an exploratory effort into analyzing code similarity using code compression, we produced a more rudimentary measure of similarity between two files. In tables 2 and 4, we present the results of our code compression tests.

In the Fibonacci Sequence code, the file that exhibited the most compression was the same file that returned a matching with MOSS and Sherlock but was not highly ranked pre-testing. The file with the second highest difference in the compression experiment was the false negative mentioned above. This file was ranked in the top two when we rated files on usefulness; this could be a case in which code compression yields better results than using the code plagiarism tools. Overall, the five files that had the largest compression differences were all ranked in the top five. This method, at least with this code sample, contains promising results for future direction.

See Tables 2 - 5 for a complete breakdown of the results.

File ID	Compressed Size (MB)
fib (base)	156
15305362	177
15515920	91
15611781	270
15820601	167
15994428	133
1678091	320
17624149	177
19871921	327
4935957	89
494594	162

Table 2: Fibonacci Sequence individual file sizes

File ID	Sum of Individual Files (MB)	Compressed Size (MB)	Difference (MB)
15305362	270	333	63
15515920	199	247	48
15611781	368	426	58
15820601	247	323	76
15994428	220	289	69
1678091	423	476	53
17624149	268	333	65
19871921	405	483	78
4935957	206	245	39
494594	245	318	73

Table 3: Fibonacci Sequence compressed file sizes

File ID	Compressed Size (MB)
hangman (base)	1344
19898080	180
6339473	492
9855011	762
9970378	581
17983653	302
19246381	1237
19341882	1174
19760186	1132

Table 4: Hangman individual file sizes

File ID	Sum of Individual Files (MB)	Compressed Size (MB)	Difference (MB)
19898080	1442	1524	82
6339473	1769	1836	67
9855011	1982	2106	124
9970378	1774	1925	151
17983653	1565	1646	81
19246381	2438	2581	143
19341882	2373	2518	145
19760186	2340	2476	136

Table 5: Hangman compressed file sizes

We believe that the code plagiarism detection tools were much more sensitive than we originally predicted them to be. We had assumed that they would match any bits of code that were similar, even in very minor ways; instead, it appears that the code has to have almost identical structure for similarity to be detected. Our overestimation of the lack of sensitivity of these tools led us to these results. As for code compression, we obtained better results that we had expected. The files that we deemed helpful all appeared in the top half of the rankings.

6 Conclusions and Future Directions

Detecting similarites in code is difficult because the current use cases are fairly specific and tools haven't yet been developed to address a more general, broader case. The tools that currently have been developed are focused on meeting the exact needs of those use cases. One use case we have found to be particularly important is detecting similarities in code that are less obvious than exact duplication. We theorize that with this broader case, a tool can be developed to allow people to find online forum posts containing code that are similar to existing code that the user has supplied because they think it is helpful. Through our experimentation, we discovered how fine-tuned the tools were for particular cases; this conclusion was reached after the tools did not detect similarities between code samples that are obviously similar.

One problem may be the granularity of some of these tools. Perhaps, they could be altered to work in our use case, potentially by either changing source code or combining them with other pre-existing tools. Future work may include research into the viability of producing a new tool that integrates several similarity detection techniques to be able to identify all four types of code clones. This research would likely involve research into heuristics that can accurately weed out many of the samples and only run detailed analyses on those most likely to be similar.

References

[1] A. Aiken. Moss: A system for detecting software plagiarism. http://www.theory.stanford.edu/ aiken/moss/.

[2] B. Baker. On finding duplication and near-duplication in large software systems. *Proc. IEEE Second Working Conference Reverse Engineering*, pages 85–95, 1995.

[3] M.H. Halstead. Natural laws controlling algorithm structure? *ACM SIGPLAN Notices*, 7(2):19–26, 1972.

[4] M.H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[5] M. Joy and M. Luck. Plagiarism in programming assignments. *IEEE Trans. Education*, 42(2):129–133, 1999.

[6] M. Mozgovoy. Desktop tools for offline plagiarism detection in computer programs. *Informatics in Education*, 5(1):97–112, 2006.

[7] J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, 1976.

[8] J. Ottenstein. A program to count operators and operands for ansi-fortran modules. *IBM Technical Report CSD-TR-196*, June, 1976.

[9] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a st of programs with jplag. *J. Universal Computer Science*, 8(11):1016–1038, 2002.

[10] Prajila Prem. A review on code clone analysis and code clone detection. *International Journal of Engineering and Innovative Technology (IJEIT)*, 2(12):43–46, 2013.

[11] S.S. Robinson and M.L. Soffa. An instructional aid for student programs. *SIGCSE Bulletin*, 12(1):118–129, 1980.

[12] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[13] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. *SIGMOD*, pages 76–85, 2003.

[14] M.J. Wise. Yap3: Improved detection of similarities in computer programs and other texts. *Proc. 27th SIGCSE Technical Symposium*, pages 130–134, 1996.

SketchyCode: An Accessible Interactive Thinking Space for Programmers.

Senior Conference Final Report

Team 08: Z. Lockett-Streiff and N. Verosky

December 15, 2013

Abstract

Although popular Integrated Development Environments (IDEs) like Eclipse gather development project resources into a central interface to streamline code management, they take traditional text editors as their point of departure and therefore generally lack tools for fluidly interacting with projects at their most abstract, structural level. Research projects like CodePad [PGR10] have explored ways of filling this gap in existing IDEs' functionality by tying birds-eye-view code visualization into the coding environment using specialized hardware, but to our knowledge no previous projects have taken advantage of tablets' accessibility and portability to build interactive code-sketching tools for mobile touchscreen devices that interact with traditional IDEs.

We present SketchyCode, an Android application that sits on top of a backend Eclipse plugin to represent software project hierarchies as collections of free-floating modules on a drawable background canvas. SketchyCode merges a two-dimensional model of Eclipse's project tree with on-screen pencil-and-paper sketching to facilitate brainstorming and project reconceptualization as part of the development process. Finally, we outline an experimental setup that could use SketchyCode to explore the potential impact of integrating high-level code sketching and visualization into traditional programming environments and discuss SketchyCode's broader possible uses in development, education, and communication.

1 Introduction

Getting bogged down in the minutiae of planning out a large-scale programming project can be messy. Furthermore, few widely-known tools exist to facilitate this process outside of the classic media of paper and whiteboards, which can become cluttered and disorganized with disjoint thoughts. Some of these existing tools require specialized and expensive hardware that may not be accessible to the average programmer.

While an IDE like Eclipse enables users to view outlines of their projects, such applications are often incredibly cluttered and create information overload. A typical IDE workflow contains the following, among other components:

- The text editor window
- A package explorer
- A project outline
- Other auxiliary functionality which may vary with the type of project being developed.

Eclipse has done well with organizing these modules to optimize space for the editor window. However, this high concentration of information may overwhelm the developer's ability to focus their thoughts. For instance, Parnin et al. cites the phenomenon of *navigation jitter* in which a developer rapidly navigates through numerous document tabs [PGR10], resulting in an accumulation of unproductive time. We propose SketchyCode, an Android tablet app which communicates with the Eclipse IDE to

provide users with a dedicated 'skeleton' perspective of their projects and enables them to sketch out their ideas.

1.1 Idea

The essential idea behind SketchyCode is to integrate high-level pencil-and-paper sketching into the development process to give the developer a way of interacting more fluidly with existing code, wrapping their understanding around a given project's overall structure, and imagining possible directions for extending and modifying current behavior. Since this birds-eye-view taking inventory of a project as it stands and considering potential structural modifications is the kind of work a programmer might want to do without being bound to a desktop computer and is not widely accessible given the specialized hardware required by systems like CodePad, SketchyCode aims to let viewers intuitively visualize changing relationships between projects' components on mobile devices. Specifically, we implemented this as an Android app that fetches a given project hierarchy from SketchyCode's backend Eclipse plugin and graphically represents the project as a collection of free-floating type-level modules on a drawable background canvas. We believe that this setup will allow developers and educators to efficiently and conveniently conceptualize and communicate programs' structural relationships without having to adapt to new programming environments or acquire specialized hardware.

1.2 Motivation

Our target audience for this tool is the broad spectrum of programmers for whom access to an Android tablet is feasible. We recognize the limitations inherent in our demographic, but restricting our scope was necessary for a project still in its prototype stage. SketchyCode grants users the ability to not only organize the components of their project in a single location, but also to draw connections between them to indicate related ideas. Equipped with a visualization of the big picture of a project, programmers can avoid losing the direction of their work and streamline the development process.

1.3 Contribution

The impetus for SketchyCode is the work of Parnin *et al.*, the developers of CodePad. CodePad provides peripheral working spaces which link to an IDE and enable developers to share notes and code with each other. However, CodePad has a noticeable impracticality: this technology requires specialized, expensive, and not widely available hardware. The idea of CodePad is very practical, but would reach a wider audience on a more mainstream platform. SketchyCode fills this void by providing the beginnings of CodePad functionality by tapping the thriving Android tablet market.

Our project is a unique contribution to existing research since mobile applications have only recently become popular. For reference, the first iteration of the Apple iPhone was released in 2007, and the iPad not until 2010, the year the CodePad paper was published. A cursory scan of Google Play revealed very little: an application titled "Android CodePad" which acts as a stand-alone code viewer. Android CodePad has no IDE integration or brainstorming space and thus is not in the domain of our project.

2 Background

The primary impetus for SketchyCode is the work of Parnin *et al*[PGR10], the developers of CodePad. CodePad provides peripheral working spaces which link to an IDE and enable developers to share notes and code with each other. The form factor on which we are modeling SketchyCode is the portable CodePad, a mid-sized tablet. However, the hardware of the portable CodePad is limited by the bulk of tablet computers leading up to and during 2010. The authors admit "a more apt device comes in the form of an iPad."[PGR10]. In the years following the 2010 SOFTVIS symposium, tablet form factors have been drastically streamlined. The authors of SketchyCode are more experienced with Android development, and so we created our project on the Android platform.

Previous research has also explored the potential benefits of reorganizing IDEs around "bubbles" or "modules" (as opposed to

windows or tabs). In particular, Code Bubbles[cod] allows programmers to group relevant project components into working sets of draggable and editable code bubbles. Importantly, the Code Bubbles team has conducted usability studies demonstrating not only that this more fluid two-dimensional layout can reduce development time but that reductions in development time stem primarily from changes in how the user conceptualizes a project and stores it in working memory – and only secondarily from more obvious differences in interface navigability and the time it takes to execute a predetermined task.[BZR+10] Of course, Code Bubbles still operates at the level of statement-for-statement code manipulation and is not concerned with integrating birds-eye-view sketching into the development process, which suggests that higher-level project organization tools using draggable bubbles might amplify Code Bubbles gains in project conceptualization, even (or especially) if they don't impact the click-for-click literal code manipulation process. Finally, Code Bubbles' usability studies suggest that sketching tools' somewhat abstract goal of streamlining the way developers understand projects and manipulate them in working memory really can lead to quantitatively verifiable changes in development time.

3 Evaluation

We successfully implemented a SketchyCode prototype that allows the user to export an existing Eclipse project to their Android device for sketching. The prototype includes an Eclipse backend that packs a given project's hierarchy of types and methods into a string that is sent over the network to SketchyCode's Android frontend. The Android frontend maps the project's types onto modules floating on a sketching-enabled background canvas and nests the project's methods within the appropriate type-level modules. The user can conceptualize the project from different angles by dragging modules to different spatial locations on the canvas, toggling modules based on relevance (or deleting them entirely), adding new candidate methods to existing types, and performing basic pencil-and-paper sketching on the background canvas.

3.1 Workflow

We will walk through the workflow of our app using a sample Java project, SCDemo. The SketchyCode workflow starts with the IDE. The user selects their project in the package explorer and then selects the "Send to SketchyCode" option in the toolbar menu (Figure 1).

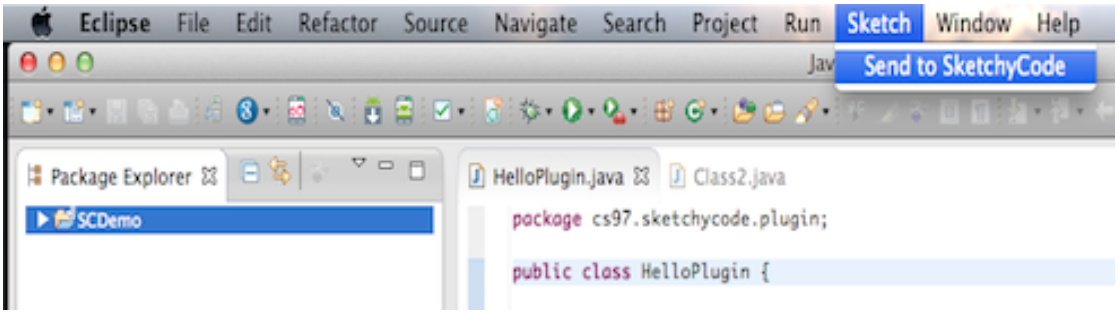

Figure 1: Sending a Project to SketchyCode

This action prompts Eclipse to set up a server to which a String containing project metadata is sent (package name, class names, method names). Upon opening SketchyCode, the app accesses the server, retrieves the String, and parses it. The components of the parsed String are passed into NoteModule. These NoteModules are displays on the tablet screen, along with functionality for writing on the screen, drag-and-drop, and adding to, collapsing, and deleting methods from modules (Figure 2). Additional ancillary functionality for erasing writing and refreshing data from the server is also available. As a weak security

mechanism, the server is closed once accessed by the tablet. Refreshing requires the user to re-send their project to SketchyCode.

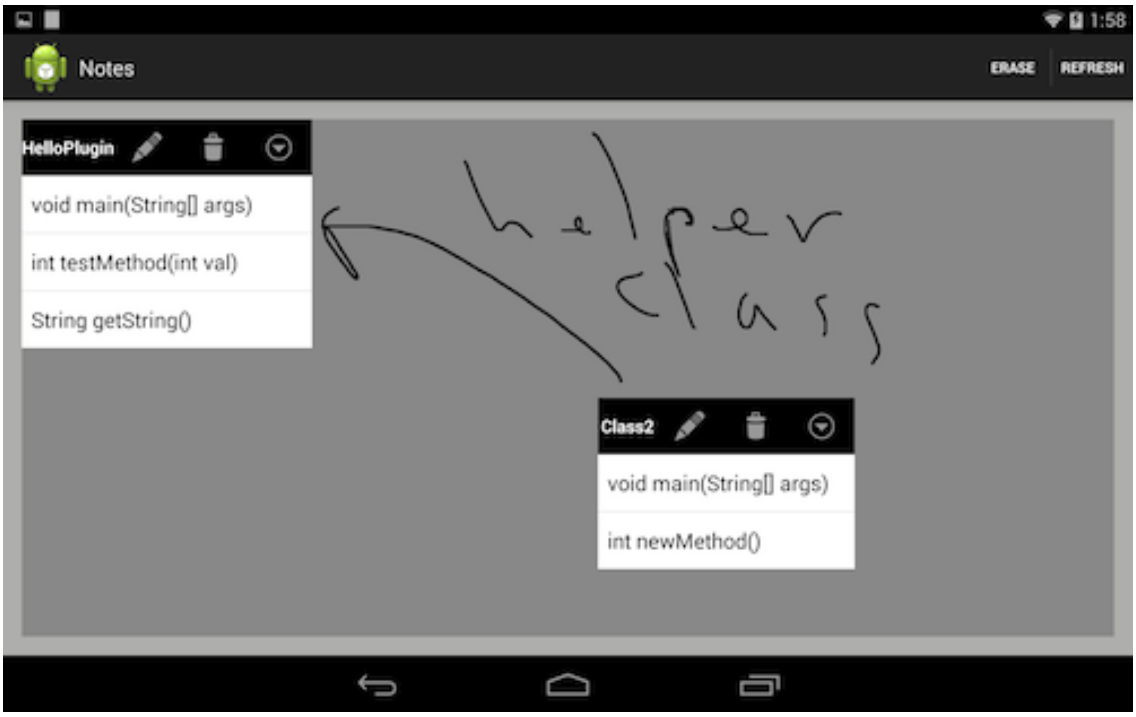

Figure 2: Inside the tablet app

3.2 Impact and Next Steps

SketchyCode has reached a stage where its ability to facilitate software conceptualization and brainstorming can now be meaningfully tested. We propose an experiment in which individual participants are each given access to a complex software project consisting of multiple interacting components and asked to make a specific change to the software's behavior with no additional documentation or explanation. Subjects would be given access to the code as an Eclipse project and would be free to use pencil, paper, and any additional development tools they deem helpful. Participants would receive a brief tutorial on basic SketchyCode and Eclipse usage, after which they would be randomly split between a control group and a SketchyCode group which would be given access to SketchyCode during the software development task.

We expect that given a sufficiently complex programming task, the SketchyCode group would be able to make the specified project changes in a shorter period of time than the control group if SketchyCode does indeed help streamline software reconceptualization. Secondarily, this experiment would give us an opportunity to systematically collect more qualitative data about how SketchyCode impacted participants' development experiences, how its interface might be made more intuitive, etc. and compare this feedback against subjects' performance on the software development task.

Although SketchyCode's stated goal of streamlining the way projects are planned, represented in working memory, and understood throughout the development process is primarily qualitative, previous studies like that conducted by Code Bubbles' developers [BZR+10] indicate that development tools' effects on high-level project organization lead to concrete gains in development time on top of more subjective improvements in ease of use and fluidity of thought. The Code Bubbles study suggests that for tools providing more intuitive

code visualization environments, times gains attributable to increased fluidity of software conceptualization significantly outweight those attributable to the greater click-by-click efficiency more advanced code editing functionality allows for. Since SketchyCode currently does not support code editing and therefore always increases click-by-click development time, the contrast between these layers is especially stark: a decreased task completion time in the experimental SketchyCode group would be telling since it would indicate that gains in high-level project conceptualization necessarily outweighed losses in click-by-click code editing efficiency.

Initially, we intended to implement limited code refactoring, enabling the user to control large portions of their project from the tablet. Code refactoring would drastically change our user testing. Currently, a user can view their project skeleton and plan out theoretical changes in their project without seeing any changes in their project. Refactoring would enable users to control large portions of their project by creating functions to encapsulate the major functionality of their work. A user's project could become significantly more modular by defining such functions up front. Our user testing would also change as a result of implementing code refactoring. Instead of, for example, gauging how quickly a user can navigate their project with the SketchyCode workflow, we can actually test a user's ability to create a new project from scratch using our app and compare it to the same project made using a traditional IDE setup.

4 Conclusion

While this app is still in its early stages, it shows promise as a project-organizational tool. It enables users to zoom out and view their projects at a high level so they can remain focused on the task at hand. Furthermore, we believe that the project-specific learning implicit in the development task motivating our app hints at SketchyCode's wider potential as a teaching and communication tool, and that variants of this experiment could explore SketchyCode's effectiveness in education and presentation environments beyond its obvious relevance to the development process.

One of the big next steps is implementing the tablet-to-IDE transmission and refactoring of code. SketchyCode will be even more useful if its outline visualization can be used to quickly add and remove functions, and possibly other project data as well. Project skeletons can be efficiently designed and created.

References

[BZR+10] Andrew Bragdon, Robert Zeleznik, Steven Reiss, Suman Karumuri, William Chang, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Jr. Joseph LaViola. Code bubbles: A working set-based interface for code understanding and maintenance. In *ACM Conference on Human Factors in Computing Systems*, 2010.

[cod] Code bubbles home page.

[PGR10] Chris Parnin, Carsten Gorg, and Spencer Rugaber. Codepade: Interactive spaces for maintaining concentration in programming environments. In *SOFTVIS*, 2010.

Likely : A Domain Specific Language for Image Processing

Jordan Cheney
Swarthmore College

Jake Weiner
Swarthmore College

Abstract

While there currently exist several image-processing languages that provide a simple syntactical structure and are quickly and efficiently executed, we believe that there is a void all these image-processing languages share: the inability to give immediate visual feedback. The image processing domain deals exclusively with images and image manipulations, and thus we feel this domain is uniquely suited to benefit from live coding - a compiling technique which allows for developers to instantaneously see the results of their code modifications. To this end, we have created Likely, a Domain Specific Language (DSL) that not only has a straightforward syntactical structure and a speedy execution, but also the novel feature of live coding.

All languages rely on a compiler and an interpreter, and Likely is no exception: it uses LLVM as its compiler, and Lua as its interpreter. We chose to use LLVM chiefly because it has a Machine Code Just in Time compiler, which offers support for live coding, the critical aspect of our language. In addition, it has well established optimization passes to increase function efficiency, and it can run on all major hardware architectures. Furthermore, we chose to use Lua because it is lightweight, efficient, and easily embeddable in C++, the language behind Likely. Consequently, by leveraging these open source libraries we created an extremely powerful language that is able to match the speed of other existing image processing languages, and thus it is able to differentiate itself from the field with its live coding environment.

1 Introduction

A domain specific language (DSL) is a programming language that is specialized to a particular application domain, which stands in contrast to more common general-purpose languages, such as C++ and Java. They are typically used to facilitate easier design and implementation of complex systems, and they have extremely specialized but powerful features that specifically apply to their targeted domain. This paper is the first to present Likely, a DSL that eases algorithm design for image processing projects by integrating both the intuitive aspect from high level languages and the powerful aspect from low level languages. In addition, Likely incorporates live coding, which gives the developer immediate visual feedback for all code modifications.

The primary goal of Likely is to simplify the creation of efficient image processing algorithms. To this end, we created a live coding environment that dramatically increases the usability of this DSL, and serves as the defining characteristic of this language. Thus, while there are other image processing DSLs that exist, none incorporate this feature that we feel is extremely important in algorithm creation and modification.

In addition, many DSLs are created for an extremely specific subdomain within image processing, and are only useful for very exclusive, unique situations. As a result, we feel that there is a need for a general-purpose image processing DSL, and thus Likely was created with this goal in mind. Any person who is working with image processing algorithms could benefit from using this language, and could gain a new perspective from the innovative live coding environment.

Likely also has many features that make it an ideal language for prototyping new image processing algorithms. Both the live coding feature and the included IDE are specifically designed to make the creation of new algorithms much easier and more intuitive for developers. Furthermore, despite the fact that many of its features were designed for prototyping, Likely can also be used to create algorithms for commercial or production level applications. This can be done in C++ by accessing the Likely standard library and using Likely functions, similarly to how a developer would use OpenCV or other available image processing libraries. Hopefully, this will

Figure 1: The Dream environment

Figure 2: A basic function in Dream

prevent developers from needing to rely on separate tools for prototyping and producing.

2 Background

Generally, algorithms for image processing are written in one of three different types of languages: high level languages (e.g. Matlab), low level languages(e.g. C, C++), or domain specific languages (e.g. Diderot), and each language type has its own distinct pros and cons. High level languages are typically used because they have simpler language structures and it is often quicker to write functional code in them. For example, Matlab has a very simple interface for matrix operations and manipulations, which allows for quick and easy image processing algorithm design. On the other hand, low level languages are used because they offer far more optimization and execution speed when dealing with large amounts of data. This is particularly valuable because image processing algorithms typically deal with massive data sets, which need to be analyzed and processed in a reasonable amount of time. Nonetheless, low level language code is normally more complex and difficult to write, and thus can be a barrier to people with less coding experience.

DSLs were created to combine the simple language structures of high level languages with the speed of low level languages. This idea has proved to be quite useful in a variety of different domains, and accordingly several DSLs for image processing already exist. However, many were written for subdomains within the image processing space and are not useful in the general case. For example, computed tomography and medical resonance imaging are used to measure a wide variety of biological and physical objects, and the increasing sophistication of imaging technology creates the demand for equally sophisticated computational techniques to analyze and visualize the image data. To this end, a domain specific language called Diderot was created specifically for this image processing subdomain, and through the creation of a high-level model of computation based on continuous tensor fields, it provides imaging scientists the ability to quickly develop reliable, robust, and efficient code [6]. Other subdomain specific examples for image processing DSLs include Teem, a light-weight collection of libraries that was designed for image processing research [4], and Scout, a data-parallel programming language for graphics processors [3].

In addition to subdomain specific DSLs, there exist several general DSLs for image processing which are comparable to Likely. Some languages, like Adobe Pixel Bender, have well documented language structures and hardware-independent compilers that perform domain specific optimizations. Pixel Bender has its own unique IDE that creates a low learning curve for users, and by supporting parallel processing and all bit depths, this DSL has proven to be extremely useful. Moreover, Pixel Bender can run on a variety of additional Adobe applications, which further increases its popularity [1]. Another example of a general DSL is Neon, which allows image processing algorithms to be written in C#, but compiles them into human readable, highly efficient, code optimized for a multicore CPU or a GPU [5]. Nonetheless, Neon was only recently created, and thus not as widely used in the image processing community. The goal of Likely is to have comparable functionality and efficiency compared to similar general purpose image processing DSL's. We hope that this, combined with the live coding feature, will be the catalyst for wide spread adoption of Likely within the image processing community.

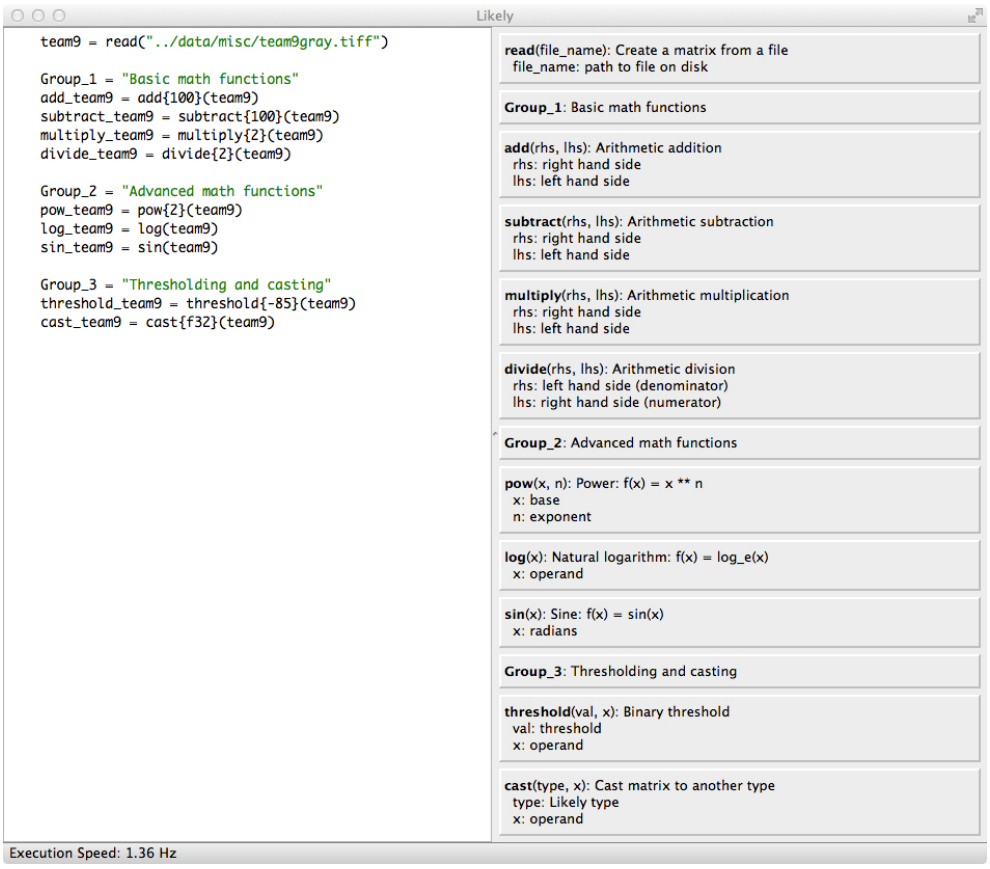

Figure 3: Examples of Likely language syntax

3 Our Idea

3.1 Technical Background

Likely was designed with three goals in mind: first, it needed to be as fast or faster than a low level language like C or C++, second, the language structure needed to be simple and intuitive to use, and third, the novel feature of live coding needed to be supported. These goals informed all of the design decisions that we as developers made. To this end, we chose to write Likely on top of the Low Level Virtual Machine (LLVM) compiler infrastructure. Using LLVM has provided several benefits: first, LLVM incorporates many powerful and well-tested optimization passes. Furthermore, in addition to these generic optimizations, we also created domain specific optimizations that we coded from scratch, and together all of these optimizations ensure that Likely functions are extremely efficient. Second, LLVM includes a Machine Code Just-In-Time (MCJIT) compiler. The MCJIT is the engine that makes the live coding environment possible. Just in time compiling works by first compiling the entire Likely standard library into LLVM's intermediate representation byte code when the IDE is launched. Next, when a user creates a new algorithm using functions from the standard library, it takes the given function inputs and inserts them into the compiled byte code of the called function. This code is then compiled to machine code and executed "just-in-time", with the results displayed in the IDE. Building on top of LLVM provides one final benefit to Likely: the ability to run on almost all architectures [2]. Likely is written in a portable subset of ISO C++, which means that it can run anywhere that LLVM can. This means that more people on more machines will be able to access and benefit from Likely.

The other goal of Likely is to provide a simple intuitive language for creating new image processing algorithms. Creating a language begins with an interpreter. In this case, Likely leverages the interpreter included with the scripting language Lua. Lua was chosen for several reasons, none being more important than the fact that it is fast, lightweight, and it is very easily embed in other programming languages, especially C++, which is of course useful in the case of Likely. The Lua interpreter and its typing system allows for user code to be quickly converted into C++ code and then into the LLVM byte code. This system also allows new domain specific functions to

	Inexperienced Programmer	Experienced Programmer
No Experience with Image Processing	1.5	3.0
Experience with Image Processing	N/A	4.0

Table 1: Results of User Testing (each number represents average level of user satisifaction with Likely on a scale of 1-5, 1 being unsatisfied, 5 being very satisfied)

be saved into the Likely standard library and later used by other developers.

3.2 The Live Coding Environment

Live coding is the single most important and unique feature of Likely. In order to implement this property, Likely comes with its own Integrated Development Environment (IDE) called Dream. Dream is divided into two sections: a code section on the left and a live section on the right. All live coding functionality in Dream is enabled by a CTRL+Click (hereafter referred to as clicking), and clicking on an image variable in the code section will cause it to appear in its most recent state in the live section, while clicking on any image in the live section will cause it to disappear. Moreover, a click in the code section on an instance of a function call will cause basic information about that function to appear in the live side of the IDE. This information includes a basic function description, required inputs, and basic input descriptions.

In addition to providing visual feedback for algorithms, the live side of Dream is also useful for error reporting. Because Likely just-in-time compiles new algorithms, as developers type errors can be detected in real time, and when detected an error message is displayed on the live side. This can be very useful for developers because real time error reporting can make errors much easier to interpret and correct, as they typically were created on the most recently changed line.

3.3 Language Structure

Within Dream, Likely has a simple language syntax for algorithm creation. Likely only has one object, the likely_matrix, which represents images or videos, and each matrix has several properties that are contained in its matrix header. Each header contains basic information about the matrix, namely the data type, the number of channels, rows, columns, and frames. Further, it also contains permissions for the matrix: whether it can run in parallel, whether it can run on the GPU, and whether or not arithmetic operations on it can become saturated. All of these values can be retrieved or set with accessor functions from within Dream. Lastly, because Likely has only one object type, there is no need to specify data types such as int, float, etc.

To modify matrix objects functions can be called. Function calls have the form $new\_matrix = function\{input\}(old\_matrix)$.... Functions are defined as either nullary (they take no inputs), unary (one input), binary (two inputs) or ternary (three inputs). A useful aspect of functions in Likely is that inputs that are known at compile time are differentiated from inputs not known until run time. Visually, braces ($\{\}$) signify a compile time argument, typically a constant value, while parentheses ($()$) signify a run time argument, typically an image. Using different identifiers for these argument types provides further opportunities for code optimizations. Additional syntax, including if statements, for loops and while loops are important language elements and are reserved for future work.

4 Evaluation

4.1 User Testing

In order to properly evaluate Likely, we set up an experiment that aimed to evaluate two distinct properties: how enjoyable and easy it is to program with Likely, and what groups of people will find Likely to be useful and would potentially employ Likely in future projects. As a result, we had multiple participants program with Likely, and after initially allowing them to explore the IDE for two minutes, we gave them multiple fundamental tasks to complete. We gave each participant a total of twenty minutes, and all but one finished in the allotted time period. After either twenty minutes had gone by or the participant completed all the assigned tasks, we gave them a five-question survey that served to measure the effectiveness of Likely. Each question was scored on a scale between 1 and 5, and we averaged the five scores in order to create a final overall score that showed the participants general satisfaction with using Likely. After this survey was completed, we also asked participants to rate both their familiarity of working with image processing, and their familiarity of writing code, as either experienced or inexperienced.

The results of our experiment exposed that general experience with programming correlated with whether a participant would find Likely to be both easy to work with and practical to use. Accordingly, participants who

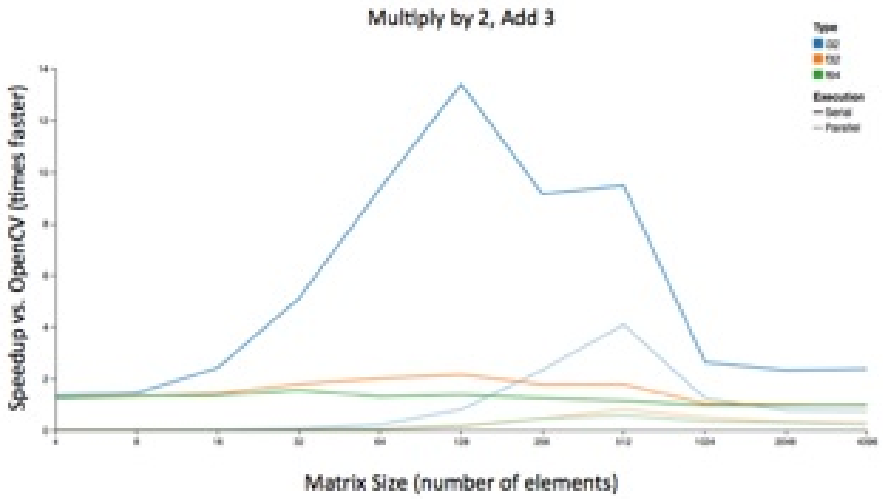

Figure 4: Likely benchmark results. Speed is calculated over OpenCV image processing library.

had no coding experience took a long time to complete tasks that other participants quickly accomplished, and thus we believe people who fall into this category should probably not be the target audience for Likely. Furthermore, the participants who had experience working with image processing and coding gave Likely the highest overall score, which indicates that people who know how to code and have worked with image processing would find Likely to be the most useful.

Nonetheless, it is important to note that only four people participated in our study, and thus this experiment has an extremely small sample size, and thus no statistical conclusions can and should be made. This was strictly a preliminary study that was created to show how one would go about measuring the effectiveness and target audience of Likely, and further data needs to be collected in order to draw valid conclusions.

4.2 Results and Performance

Likely now has a completed backend structure that supports fast function execution, as well as a live coding environment with a basic language structure. Nonetheless, the Likely standard library is still incomplete, and thus it is imperative that new functionality is added to ensure that Likely can be useful as a DSL. To this end, we have implemented functions that help to achieve this goal, and we have added functionality for several basic image processing functions. Likely now supports ten data types, unsigned 8, 16, 32, and 64 bit, signed 8, 16, 32, and 64 bit and floating point 32 and 64 bit. Additionally, casting between all of these data types is possible and is done internally for several functions. Thresholding images at user specified threshold values are now supported as well. Expanding the standard library continues to be the priority for the language and represents a large part of future work.

As was mentioned in the idea section, for Likely to be worthwhile and practical it needs to be as fast or faster then generic C++. To this end, a benchmark test was written for all functions implemented in Likely. The comparison function was written in OpenCV, a collection of open source image processing libraries written in C++ that is popular within the image processing community. The results for the multiply and then add (madd) function are shown in figure 3.

5 Conclusion

While the creation of Likely is off to a good start, there still is a lot of work to be done before Likely is considered to be a viable alternative to existing DSLs. Mainly, the Likely standard library needs to be expanded until it reaches the standard set by other open source image processing languages. For Likely to ever be widely adopted, this goal is of the utmost importance, and thus achieving this is still our number one priority. We also hope that the live coding feature promotes a paradigm shift in how developers think about writing code, and we hope that this shift occurs not just in image processing, but in other applicable domains as well. While Likely still has a long way to go before it meets the industry standard, we believe that there is a strong foundation on which to build, and we are excited to see how Likely will evolve in the upcoming years.

In summary, the ultimate goal of Likely is to give developers the ability to easily create efficient image processing algorithms. As a result, Likely incorporated an

innovative live coding environment and included built in efficient domain specific optimizations. We hope that these contributions will facilitate future research in image processing algorithms in the academic community. In addition, the user testing that we performed indicated that people with image processing and programming experience had the best experience using Likely, and furthermore these types of people indicated that they see real value in using Likely. We hope that Likely can find traction outside of the academic community, and we hope that the live coding environment and simple language structure will make Likely an attractive option for regular users who want to create image processing algorithms.

References

[1] Adobe. *Pixel Bender Developer's Guide*.

[2] Edward Barrett. 3c - a JIT compiler with LLVM. Technical report, Bournemouth University, 2009.

[3] P. McCormick J. Inman J. Ahrens J. Mohd-Yusof G. Roth S. Cummins. Scout: A data-parallel programming language for graphics processors. *Journal of Parallel Computing*, November 2007.

[4] Gordon Kindlmann. Teem library.

[5] Brian Guenter Diego Nehab. Neon: A domain-specific programming language for image processing. Technical report, Microsoft Research, 2010.

[6] Charisee Chiw Gordon Kindlmann John Reppy Lamont Samuels Nick Seltzer. Diderot: A parallel dsl for image analysis and visualization. Technical report, University of Chicago, 2012.

Identifying Potential Concurrency Bugs by Analyzing Thread Behavior

Stella Cho (scho1) and Elliot Weiser (eweiser1)

December 20, 2013

Abstract

Multithreading is one of the most prominent approaches to concurrency in modern applications. The benefits of multithreading include increased performance, modularity, responsiveness, and interactivity. However, concurrency bugs have made debugging and testing multithreaded software extremely difficult. We claim that the likelihood of concurrency bugs in multithreaded software depends in large part on the behavior of the threads. In this paper, we conduct a measurement study on commonly used multithreaded, Mac OS X software in order to classify their thread behavior according to the following two behavior types: *Computation* threads and *event-handler* threads. We expect event-handler threads are, due to their short and unpredictable behavior, more likely to contain concurrency bugs. Overall, a better understanding of thread behavior can inform future efforts to eliminate concurrency bugs.

1 Introduction

One of the most highly researched topics in computer science is concurrency, which is the simultaneous execution of several tasks on a computer system. Simultaneity can be either physical or simulated: physical simultaneity is achieved by executing different tasks on different processors, whereas simulated simultaneity comes from interleaving the execution of different tasks on a shared processor space. In either of these cases, it is possible for these tasks to interfere with each other in negative ways [6].

The rising need for concurrency can be traced to two key factors: increased computational demands and advancing hardware. Modern software must perform computationally intensive tasks, such as scientific computing, image and video processing, and running web servers. Supporting software of this kind requires more sophisticated hardware. With Moore's Law set to expire, chip manufacturers have adapted by developing multi-core technologies, such as multiprocessors, GPUs, and FPGAs. Several software-level concurrency abstractions have arisen to drive this advancing hardware and to more fully take advantage of a computer's architecture.

Multithreading is one of the more common approaches to single-system concurrency. A thread is a preemptive, OS-managed, lightweight process. In the best case, threads can achieve physical parallelism on multiple processors, and thus achieve higher performance. When the OS needs to make scheduling decisions, simulated simultaneity takes over. Compared to processes, the primary software abstraction for an OS-managed instance of a running program, threads are more lightweight because they share their parent's memory and address space. By separating concerns among threads, multithreading can provide modularity. Threads can also help increase the responsiveness and interactivity of a program. For example, a thread can wait around for user input.

The major risks associated with multithreaded software arise from the sharing of resources. Sharing memory with the parent increases the potential for race conditions. Programs can

deadlock when two or more threads are waiting for the other to finish. Thread starvation can occur when a thread runs indefinitely without relinquishing some shared resource, preventing other threads from making progress. As a great deal of time and money has been spent on testing, debugging, and researching new ways to avoid concurrency bugs, developers would benefit from research into reliable multithreading. Further, because multithreaded software is virtually ubiquitous, research into minimizing these risks has the potential to affect everyone who owns and operates a computer.

We claim that the likelihood of multithreaded software to contain concurrency bugs is linked to how its threads behave. Certain thread behaviors, we believe, are riskier than others. In this paper, we present our measurement study of thread behavior in multithreaded software systems. In Section 2, we discuss earlier research that address problems associated with multithreading. In Section 3, we highlight the primary objective of our measurement study, define two thread behavior types, and describe the methods for our data collection and analysis. In Section 4, we present the results of our experiments for different pieces of multithreaded software on Mac OS X. We present the implications of our experiments in Section 5, as well as future directions to take this research, and our concluding remarks.

2 Background & Related Work

Concurrency bugs in multithreaded software have plagued researchers and developers for decades. Various efforts to reduce thread-level bugginess have been introduced, such as Stable Multithreading (StableMT). StableMT attempts to make multithreaded software more reliable by mapping inputs to a restricted set of schedules that have been tested for correctness [5]. Other approaches to multithreading have evolved, such as cooperative threading, which lowers the probability of race conditions by running only one thread at a time, and yielding only given an explicit yield invocation [6]. These efforts, however, are far from perfect solutions. For example, cooperatively threaded systems are prone to thread starvation.

We expect that a better understanding of thread behavior will advance efforts to reduce thread-level bugginess. In order to characterize the behavior of a given thread, we need to be able to study its history. Recent work by Trümper *et al.* (2010) and Blake *et al.* (2010) provide some of the tools necessary for studying thread behavior, although for motivations different from those expressed in this study. Trümper *et al.* develop a tool for understanding multithreaded software behavior more holistically, for the purposes of software optimization and understanding system behavior. Their visualization software allows users to select a representative subset of spawned threads and trace their method boundaries [4]. Our study is more interested in the behavior of the threads themselves. Blake *et al.* study how effectively modern desktop applications use threads on multi-core architectures by looking at context switches and GPU utilization, but focus mainly on CPU *idle time* and do not collect statistics on thread behavior [1].

We base our method on Blake *et al.*'s experimental setup. In particular, Blake *et al.* use *DTrace*, a dynamic tracing framework that can report when a thread is created, destroyed, started, or stopped [3]. Developed by Sun Microsystems, Inc., *DTrace* was originally created to help developers observe, debug, and tune system behavior, and is currently available for Solaris, Mac OS X, FreeBSD, NetBSD and Oracle Linux.

3 Our Idea

3.1 Objective

The more we can say about a thread's behavior, the more we can assess its risk to the software. Our objective is to measure several pieces of commonly used software in order to classify their thread behavior according to the two following behavior types: *Computation* threads and *event-handler* threads.

3.2 Definitions

Computation threads satisfy the conventional need for parallelism; they are often used to divide up equal amounts of disjoint work across several processors (see **Figure 1**). The optimum is to use roughly as many threads as there are processors, otherwise time is wasted dealing with context switches, OS-communication, and cache misses.

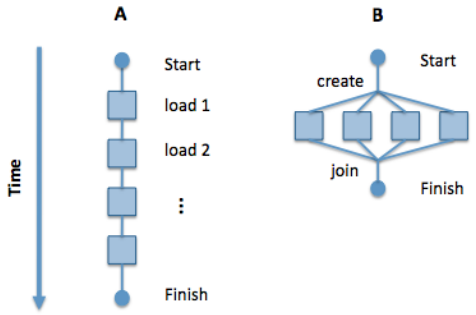

Figure 1: Computation threads can increase performance of a non-parallelized program (A) by doing similar-sized chunks of work at once (B).

Consider a classic, parallelized matrix multiplication algorithm. If the input matrices grow, then the threads will be doing either more decimal multiplications or more dot products. Note that the runtime of a computation thread grows with respect to the size of its input. Furthermore, we expect that computation threads have very active lives, by which we mean that the ratio of the time a thread is on a CPU doing work to the time that it is alive (i.e. created, until destroyed) is very high. The number of context switches occurring on that shared processor space has the potential to deflate this ratio.

Event-handlers are routines that execute upon the realization that some "event" has occurred (i.e. they are told to execute when some condition has been met; see **Figure 2**). Traditionally, these are run synchronously as part of an event-loop and are invoked by a dispatcher. However, threads can handle events as well, eliminating the need for a dispatcher.

Consider the following implementation of a

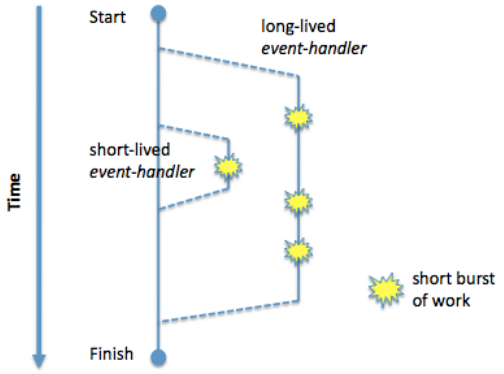

Figure 2: Event-handlers do very short bursts of work. An event-handler's lifespan can last anywhere from microseconds to the entire lifespan of the program.

web server: when a listening socket gets a new connection, it spawns a thread upon receiving a new request. The thread processes the request, sends the appropriate response message, closes the connection, and exits. In this instance we see that input size has little to no bearing on a given thread's runtime. Furthermore, we expect that event-handlers either have short lifespans or that they work in short bursts before transitioning to some inactive state (off the CPU; see **Figure 2**). Some event-handlers may have such short lives that they are on the processor once and are destroyed before they have the time to context-switch.

The problem with event-handlers is that they have more unpredictable behavior; we cannot always know when they will be triggered, and if they are, they can have detrimental effects on the rest of the software [6].

3.3 Methods

Using *DTrace*, we compile thread histories for various multithreaded software systems. We analyze these thread histories to describe the overall behavior of threads in the system.

We studied the behavior of dozens of thread-related *DTrace* probes and identified two that gave us the most consistent and relevant results: *on-cpu* and *off-cpu*, from the *sched* provider. As

the name suggests, these probes tell us when the CPU is about to start or end the execution of a thread [3]. All of our experiments were run on an early-2011 MacBook Pro running Mac OS X 10.7.5 with a 2.3 GHz Intel Core i5 processor.

To determine the time from thread creation to thread destruction, which we refer to as the thread's *lifespan*, we measure the time from the thread's very first event to its very last. To determine *active* time, or the amount of time a given thread is actually on the CPU, we add up the time from every *on-cpu* to *off-cpu*.

If we find that thread execution time grows with respect to the input size, and the thread has a high activity to lifespan ratio, then we claim that the software in question is using computation threads. Alternatively, if the activity to lifespan ratio is very low and thread execution time does not grow with respect to input size, we believe the software is using event-handler threads.

4 Evaluation

We collected data on the thread histories of three popular Mac OS X desktop applications: Microsoft Word, Safari, and Preview. We also collect data on a Parallelized Game of Life (PGoL) program to test whether we see the behavior we would expect from a computation thread-based program. We visualized the results by graphing thread activity over lifespan against thread lifespan.

4.1 Preliminary Result

In order to test our hypothesis on computation thread behavior, we collected thread history data on PGoL. PGoL takes as input an n x n board size, specified by the user, divides the board into t segments, where t is the number of threads also specified by the user, and runs the game in parallel. We run PGoL three times with four threads and three different board sizes (n = 1000, 3000, and 5000). As we would expect from a computation thread-based program, we get the following results (see **Figure 3**):

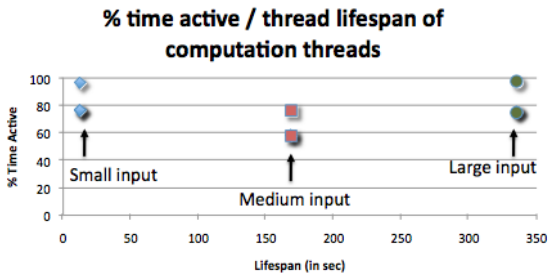

Figure 3: Graph depicting the expected behavior of computational threads, obtained by running PGol. Each color is a cluster of four threads that denotes a separate run.

1. *Clustering of thread behavior.* Because the disjoint work is evenly distributed across four threads, we see that for each run the threads exhibit roughly equal runtimes and ratios of activity to lifespan.

2. *Threads are active for a significant portion of their lives.* We expect computation threads to have very active lives.

3. *Thread lifespan is dependent on input size.* As the input size grows, so does the thread lifespan. Input size horizontally shifts the cluster.

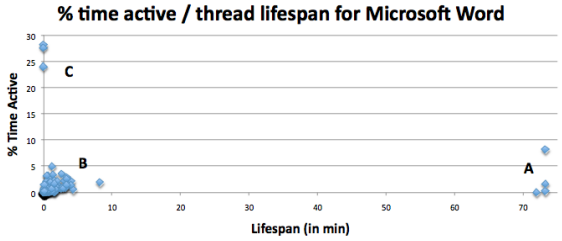

Figure 4: Graph depicting thread behavior of Microsoft Word. (A) Threads that are alive from launch to exit. (B) Threads that live a short time and are active for an even shorter time. (C) Extremely short-lived threads with a high active to alive ratio.

Next, we collected data on Microsoft Word. A 73 minute generic run of Microsoft Word, which

involved a user typing, highlighting text, and inserting comments, generated 236 threads and gave the following results (see **Figure 4**):

1. *Four threads appear to be alive from launch to exit.* However, they still exhibit what we expect to be event-handler-like behavior because they are active for a relatively short percentage of their lives.

2. *Most threads live a short time and are active for an even shorter amount of time.*

3. *Some threads are so short-lived that they have a high active to alive ratio.* As the threads begin and end almost instantaneously, they do not get context-switched.

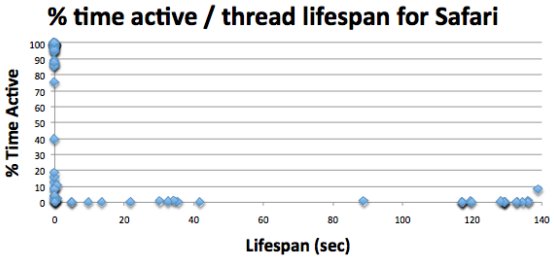

Figure 5: Graph depicting thread behavior of Safari.

These results lead us to believe that Word is exhibiting mostly event-handler-like behavior. We see similar clustering results for Safari. A 2.5 minute generic run of Safari, wherein we navigated to Google and searched for cats, yielded 100 threads. These threads fit into the categories of behavior outlined above (see **Figure 5**). We see similar results from Preview.

In order to test our hypothesis that event-handler threads are not dependent on input size, we collected data on a Preview run with a large pdf (1.8 MB) and a small pdf (11 KB). As expected, thread lifespan does not grow with input size (see **Figure 6**).

4.2 Discussion

We see from our results that three distinct pieces of software (Microsoft Word, Preview, and Safari) exhibit similar clusterings of thread behavior. The threads can be categorized into the following behaviors: threads that wait around for user input, and threads that pop up, do some activity, and disappear. Both of these behaviors are expected of event-handler type threads (see **Figure 2**). Further, we expect these software to be event-handler thread-based programs because they are highly interactive programs that are mostly driven by user input. PGoL, on the other hand, demonstrates the behavior we would expect from computation threads. That clearly is not the behavior we are seeing from the threads generated by Word, Preview, and Safari.

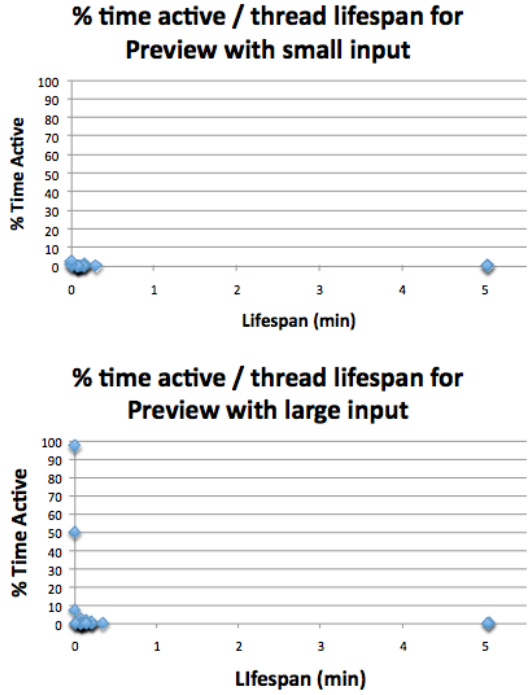

Figure 6: Graph depicting thread behavior of Preview given (above) a small input size, and (below) a large input size.

5 Conclusion

Multithreading is everywhere. Our measurement study seeks to advance efforts to minimize the potential for concurrency bugs so that we can continue to enjoy the benefits of multithreading without the risks. If we find that threads with

event-handling behavior are more likely to lead to concurrency bugs, we can adopt better approaches to dealing with concurrency bugs.

For example, Charcoal is an approach to cooperative threading that is currently in development. It is a C dialect which introduces an *activity*, which improves on existing cooperative implementations by allowing for implicit yields, thereby preventing thread starvation [6]. Activities are a possible solution to the short-lived, unpredictable behavior of event-handler threads. Further study into the thread-type composition of multithreaded software will inform the usefulness of such an activity-centric implementation.

5.1 Future Work

The *DTrace* probes we used only provide a subset of the total useable information to create a detailed description of thread histories. As future work, we would like to test and use other probes to capture blocking, mutex locking, barrier or condition waiting, and yielding. There are hundreds of relevant probes, so we have a few possible directions to take this research.

Our definitions for computation threads and event-handler threads applies more to thread behavior on an individual basis. With more information, we may be able to classify entire groups of threads. For example, we may be able to determine if a software system is using a thread pool, or perhaps employing cooperative threading. The data we are already collecting provides a promising segue into determining this kind of data.

We also do not have substantial data from software that use computation threads. We would like to experiment with other pieces of software for which we have not seen the code, and verify our PGoL results. Computationally intensive applications would be an ideal place to look.

Finally, we would like to perform more rigorous testing of the software studied in this paper, perhaps by automating the application runs. This would produce more reproducible data and allow us to form more in-depth comparisons between two runs of the same application.

References

[1] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. *ISCA'10*, 2010.

[2] Kristian Flautner. Thread-level parallelism of desktop applications. In *Workshop on Multi-threaded Execution, Architecture, and Compilation*, 2000.

[3] Solaris dynamic tracing guide. `docs.oracle.com/cd/E19253-01/817-6223/`, 2010.

[4] Jonas Trümper, Johannes Bohnet, and Jürgen Döllner. Understanding complex multithreaded software systems by using trace visualization. *SOFTVIS'10*, pages 133–142, 2010.

[5] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. Determinism is not enough: Making parallel programming reliable with stable multithreading. *smtcacm*, 2013.

[6] Ben Ylvisaker. Conventional concurrency primitives/frameworks. `http://charcoal-lang.org/big_four.html`, 2013.

BatTrace: Android battery performance testing via system call tracing

Yeayeun Park
ypark2@swarthmore.edu

Mark Serrano
mserran2@swarthmore.edu

Craig Pentrack
cpentra1@swarthmore.edu

Abstract

The increasing number of tasks we can perform on our mobile devices feeds positively into the demand for devices with longer battery power. Since mobile devices have become integral parts of our daily lives with the number of tasks we can accomplish far outpacing battery performance improvements, consumers have increasingly encountered the issue of efficient device usage and battery life management. In this paper, we examine Android devices in particular and present BatTrace, an Android analysis tool that evaluates battery performance on the android platform by tracing system calls. BatTrace will execute different types of popular system calls, and extract the correlation between a particular system call and its influence on the battery. Subsequently, it will trace system calls made by individual Android applications and use system call performance data to profile each application. Finally, the analysis on the correlation between system calls and their battery usage, as well as the correlation between each application and system calls they initiate, will be combined to estimate battery usage of individual Android applications.

1 Introduction

Our project is motivated by an issue that we face daily: limited battery power on our mobile devices. The vast power available at our fingertips in mobile devices is tamed by the amount of battery physically available. Wanting to track application behavior and the resultant energy usage, we used *strace* in combination with C programs and Android system data to perform dynamic analysis on Android devices, uncovering low-level explanations as to what is really draining the battery.

Hypothesizing system calls to be the key to understanding battery usage at a low-level, we used *strace* to profile applications system call usage. Such testing gave us aggregate statistics on which system calls were used and how often. In the process of tracing popular 3rd party applications (e.g. Facebook, Gmail, etc.), we also recorded battery usage by utilizing built in Android system data pre and post traces.

Once aware of the most used system calls, we created C programs to repeatedly run those calls and collect battery statistics. It is important to note that in future versions of BatTrace we will need to collect statistics on all system calls, as the most frequent calls may not drain the most power. However, starting with the most frequent system calls and the C programs repeatedly ran, we acquired an avetage measure of how much battery each system call consumes. From the aggregate trace data and average system call consumption data, we were able to make predictions on battery usage of 3rd party applications. In collecting high level data on battery con-

sumption, we hope to better inform Android users which applications drain their battery level most, providing a good set of guidelines for mobile users to follow when low battery crisises hit. With our fine-grained system call data, we hope to better inform software developers and hardware manufacturers which system calls consume the most power.

2 Background and related work

Historically much power consumption research has focused on using utilization-based methods (on individual components, e.g. CPU). However, modern smartphones employ complex power strategies in device drivers and OS-level power management, sometimes rendering utilization as a poor model for representing power states and deducing battery usage (**?**). While sometimes strong correlation exists between utilization and power consumption, often applications have constant power consumption while in certain states (while utilization fluctuates) or have high power consumption while low utilization (**?**; **?**). Merely focusing on CPU and other component-specific utilization levels do not capture tail-power states and the more intricate workings of power management. Additionally, measuring utilization via performance counters results in accuracy loss (**?**).

Instead of modeling power with utilization, system calls, the only way of interacting with hardware and performing I/O, serve as a much more precise indicator of power consumption (**?**). System calls, an indispensible aspect of mobile applications, provide accessible insight in how an application is using the underlying hardware. Past work and tools, most notably eProf, show correlating application behavior with power usage (via system calls and power readings) has been more successful than utilization-based approaches (**?**; **?**; **?**; **?**). Using the findings of eProf and other studies as justification, we measured and classified system calls on Android smartphones in terms of their effect on battery life.

While eProf foregrounded system calls as an effective indicator of changes in power state, eProf used system calls as a means toward profiling applications' power consumption on a sub-routine level (**?**; **?**). Developing models based off of system calls supplied a powerful tool, however the eProf research did not study battery drain as a result of particular system calls themselves and the frequency with which applications rely on certain system calls. Other work in smartphone battery research, including detecting energy-related bugs, correlating wireless signal strength with battery consumption, and generating battery usage information on the process or application level, has relied on system calls (**?**; **?**; **?**). We plan to supplement the research area by focusing our study on the system calls themselves, rather than using them as a means in tracking changes in power state, detecting bugs, measuring signal strength effects, or producing higher-level profilings as explored previously.

3 Our Idea

While dynamic analysis on traditional devices involves the most efficient use of finite computing resources, mobile devices introduce a new problem; finite power. The issue we immediately encounter when trying to analyze mobile software applications is that we ~~almost~~ never have access to the source code of the applications. This is especially true given the fact that most mobile software is proprietary in nature, leaving open source software to the relics that are desktop computers.

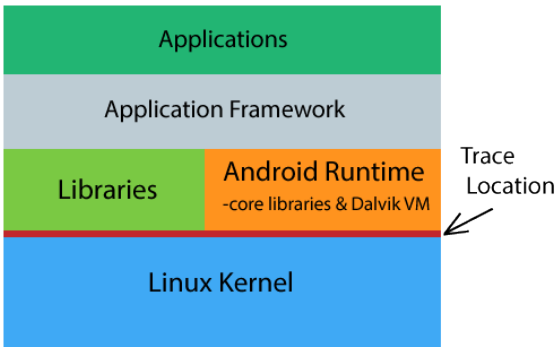

Figure 1: The Android environment stack. Trace location marks where we will be intercepting system calls

With this in mind we set out find a way of measuring mobile battery usage at very low level (software wise). We decided a good approach would involve monitoring activity at the system call level using *strace*. We wanted to profile a variety of system calls based on how much battery is used while they are running. We intended to establish a baseline

battery consumption level so we know how much battery is used by just the OS. Then, using simple programs that repeatedly make the same system call many times, as a single call is often immeasurable, we can determine how much battery was used as a result of initiating a particular system call.

Once system calls were profiled, we proceeded to the last phase of the analysis. Our goal was to identify the system calls initiated by the Dalvik VM as a result of running an individual app. By identifying the types of system calls, as well as the number of calls made to an individual system call, we were able to predict applications' impact on the battery based on what we learned about battery usage for individual system calls. While this approach may not be the most accurate, we believe it is an approach that will allow us to profile any application regardless of the author or the nature of the software's license. While this design is not the only approach, with alternatives such as dynamic binary instrumentation, we think our way capturing system calls supplies the necessary information to properly model power usage by individual Android applications.

4 Evaluation

In the process of analyzing battery usage through system call tracing we successfully found most used system calls by popular Android applications, wrote C programs to repeatedly execute those system calls, recorded battery usage as a result of particular system calls, gathered system call data of popular apps dynamically, and made predictions based on the aforementioned data. This section is split into subsections based on the above categories.

4.1 Frequent System Calls

In relating system calls to battery usage, we first traced 10 popular Android applications to acquire a list of the most frequent system calls. We traced Google Maps, Facebook, Angry Birds, Youtube, Google Play, Gmail, Twitter, Skype, Pandora, and Instagram using *strace*, collecting the ten most used system calls by each. Applications were run for at least 2 minutes simulating normal usage patters. More detailed information on the usage patterns appears in the Appendix with the system call frequency results appearing in the Table 4.1.1.

We found there were 14 most frequent, unique system calls among the applications tested. Some of the systems calls in the 14 did not make the top ten as often (and munmap never), however these calls if not in the top ten often fell just outside it, so we decided to include them. With this list, we knew some significant system calls to recreate in order to gather battery usage data. A complete test should include all of the systems calls run, however we focused on the most frequent calls in this early stage of our research. Later, including all system calls will be important as the frequent calls are not necessarily the system calls responsible for using the most power.

System Call	# Appearances
clock_gettime	10
ioctl	10
getpid	10
epoll_wait	10
getuid32	10
futex	10
mprotect	10
cacheflush	9
read	7
write	6
sigprocmask	4
gettid	2
gettimeofday	1
mmap2	1
munmap	0

Table 4.1.1 Frequent System Calls in Popular Android Applications. Appearances denote how many times a certain system call was in a top 10 most frequent system call list for each of 10 different applications tested.

4.2 System Call C programs

In developing averages for battery usage per system call, we created C programs for each of the 14 unique system calls. Once these programs were created, we ran a bash script running directly on the device that took a battery reading before and after each program completed. An example system call C program can be seen in the Appendix.

While the execution time and number of calls made varied among programs, they all ran for at least ten minutes, giving us a significant change in

battery. After running the script for each program, we recorded an average of battery level consumption per system call for each of the 14 unique most frequent. See results in Table 4.2.1.

System Call	# Battery Consumption
clock_gettime	0.006425
ioctl	0.011998
getpid	0.004161
epoll_wait	0.009926
getuid32	0.006628
futex	0.005474
mprotect	0.006501
cacheflush	0.004312
read	0.010881
write	0.016212
sigprocmask	0.005274
gettid	0.003000
gettimeofday	0.008013
mmap2	0.006066
munmap	0.006474

Table 4.2.1 Battery Consumption of System Calls. For readability consumption is displayed per every 1,000,000 calls.

Even at the level of a million calls, the battery consumption average for each system call was very small. Such small battery drain of most calls made us critical of our experimental design, measuring the average drain of a particular system call by repeatedly running it. Futhermore, each unique system call was rerun using the same parameters, while the parameters of system calls made by 3rd party applications varied. In the future, we plan to capture the battery drain of system calls in the wild, considering parameters. A more detailed discussion of how we modeled power and future improvements continues in the analysis of error and future work section. While being wary of our battery consumption averages for the 14 system calls tested, realizing our method of modeling power needs revision, we continued with our predictions.

4.3 Prediction Data

To make our predictions on battery usage of popular Android applications based off of battery consumption per call statistics, we needed a complete list of system calls made by those applications, in particular at this stage getting the number of times each system call was made. Using *strace*, we attached to application-specific processes and recorded the necessary data. Before and after attaching to processes, we recorded the battery level of the Android device to enable later comparison between predicted consumptionand observed battery consumption. The Android applications we used for predictions included Google Maps, Facebook, Youtube, and Pandora. Observed consumption data can be found in Table 4.4.1.

4.4 Predictions

With prediction data (which systems calls are made by certain applications with their frequencies) and average consumption per call for the 14 system calls we focused on, predicting battery usage was a matter of summing the estimated consumption for each system call. We predicted each system call's power usage by multiplying the number of times it was called by our estimated usage for that particular call. As we did not consider parameters at this stage in our research, the estimated battery usage for a call was the same for each call within an application and across applications. We admit such oversimplication. Once taking this sum, we added a baseline reading of how much power is used by tracing an almost completely idle application (we used weather clock) to simulate power used by the system, enabling accurate comparison to the traced 3rd party applications.

When calculating these predictions, we only had the ability to include consumption of system calls for which we had average consumption statistics. While this is not complete and may seem limiting, the 14 unique calls we focused on were the top 14 used system calls for each application tested. Comparison between predicted consumption and observed consumpted can be seen below in Table 4.4.1.

Application	Predicted	Observed
Google Maps	2	6
Facebook	2	4
Youtube	2	5
Pandora	2	4

Table 4.4.1 Battery Consumption Predictions and Observations.

After seeing the differences between predicted and observed battery consumption for the applications tested, our suspicions surrounding the way we

gathered average battery consumption per system call were made stronger. The predictions, due to the very small estimated drain of each system call, were equal to the baseline measure of tracing the mostly idle weather clock. The predicted consumption as a result of system calls was insignificant. We explore our sources of error and directions of future work in the next section, outlining how our predictions can be improved.

4.5 Analysis of Error & Future Work

Our project turned out to have had inherent flaws in the experimental design that naturally led to errors– the difference between the predicted values and the actual readings, as can be witnessed from the Table 4.4.1. We have identified three main issues that we discuss in this section:

- Lack of system call parameters: Parameters are passed to system calls in the same way that they are passed to other functions. These parameters contain important information that guides a system calls behavior. Same system calls with different parameters are essentially carrying out different tasks. In the C programs that we wrote (example shown in Appendix), we ran system calls with generic parameters that do not necessarily match those that were actually called in the test applications. In other words, it is possible and likely that system calls with varying parameters will drain battery differently. Thus, error existed in creating a average consumption for each call based on a single set of parameters while not considering the parameters of test applications.

- Insufficient data collection: Our predictions are calculations of data from 14 most frequent system calls instead of all the system calls initiated by the test applications. This design fails to capture the power consumption of less frequent system calls, and in particular less frequent system calls that use relatively high power. Due to this limitation, the power usage data we managed to collect from the top 14 system calls was not fully indicative of the actual power usage.

- Design of our C programs: We designed our C programs so that each runs a single system call repeatedly in a for loop for a set number of iterations. Though this was a clever manipulation of the knowledge we were given, we concluded that it might not have been the best design. For instance, when "getpid" is called the first time, the value returned from calling the system call is almost certainly cached. Thus, for the rest of the iterations, power usage is significantly diminished, giving lower power usage than would occur in the wild where the sequence of system calls would differ.

  In the future, we would like to improve upon these errors, making our experimental design more accurate and complete. Furthermore, we hope to expand our project, so that it can be used as a useful guide for battery-conscious smartphone users. In particular, we hope to make an downloadable tool that provides online analysis.

5 Conclusions

Despite the success of system call tracking on the mobile device, it is clear system calls alone do not provide an absolute measure of power consumption. Though we discuss some of the possible reasons why the predictions of our model were so far off, it is unlikely that these reasons are solely responsible for the differences between the predicted values and the actual battery readings. During our testing, it became apparent to us that additional power drain was occurring as a result of higher level processes that do not interact directly with the underlying OS. We suspect the Dalvik VM (and the Java code running on it) contributed to the power drain observed. However, while system calls do not reliably provide an absolute measure, they proved to be useful in calculating a relative measure of power efficiency across applications. By running a variety of apps for a set amount of time, we were able to measure power consumption via the system calls they triggered and, in turn, use the power consumption data to compare the applications to each other. A tool such as the one proposed in this paper could one day be part of the off-line development suites of software writers across the globe as the focus on energy efficiency shifts from hardware to software. However, this tool could also prove useful as an online

tool in the hands of the consumer as consumers become more aware of power hungry applications and their effects on battery life. While we were unable to develop a solid measure of battery drain using the proposed model, the approach still shows promise and our work provides a solid foundation on which to develop additional research.

References

Ning Ding, Daniel Wagner, Xiaomeng Chen, Abhinav Pathak, Y. Charlie Hu, and Andrew Rice. 2013. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In SIGMETRICS '13 Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems.

Inc. Google. 2013. Power profiles for android. Developer Guides.

Abhinav Pathak, Paramvir Bahl, Y. Charlie Hu, Ming Zhang, and Yi-Min Wang. 2011. Fine-grained power modeling for smartphones using system call tracing. In EuroSys '11 Proceedings of the sixth conference on Computer systems.

Abinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. 2012. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In MobiSys '12 Proceedings of the 10th international conference on Mobile systems, applications, and services.

Abhinav Pathtak, Y. Charlie Hu, and Ming Zhang. 2012. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In EuroSys '12 Proceedings of the 7th ACM european conference on Computer Systems.

Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. 2012. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In USENIX ATC'12 Proceedings of the 2012 USENIX conference on Annual Technical Conference.

A Appendix

A.1 Tasks Done When Tracing

When we traced popular Android applications – Google Maps, Facebook, Angry Birds, Youtube, Google Play, Gmail, Twitter, Skype, Pandora, and Instagram – in determining which C programs to recreate and in collecting prediction data we tried our best to simulate normal usage patterns. When using Google Maps we asked for directions, explored the map, went into street view, and looked for places to eat. For Facebook we browsed the Newsfeed, liked posts, wrote a new post, and looked at friends' pages. Below is a list of the tasks performed for each of the test applications when getting the most frequent system calls. When getting prediction data, the tasks were much the same however carried out over a longer period of time (at 10 minutes instead of 2 minutes).

- Google Maps: getting directions, moved around map zooming in and out, changing from the grid map to satellite imaging (street view), looked at location specific details
- Facebook: browsed Newsfeed deep into endless scroll, interacted with posts in Newsfeed, authored new post, went to friends' profiles and interacted with them
- Angry Birds: completed the first 2 levels
- Youtube: searched for 2 videos playing segments of both, played segment of video from featured videos
- Google Play: browsed popular applications (free and paid), downloaded twitter
- Gmail: checked for new mail, composed new email, went into email history and read
- Twitter: scrolled through personalized feed, searched for hashtags, clicked links to other users
- Skype: entered into video chat, did some text chatting

- Pandora: created new station, added variety to existing stations, listening to different stations throughout
- Instagram: scrolled through dash, looked through trending posts, liked several posts, looked at user profiles

While simulating normal usage seemed important in capturing averages for system call battery drain and the most frequent calls, future study should focus on usage patterns that generate a complete list of system calls with varying parameters.

A.2 Example C Program

When desigining tests to estimate the average consumption of the 14 unique system calls we considered, we created C programs to repeatedly run the call, collecting battery data before and after via a bash script. The basic structure of the C programs followed the example below.

```
/* getclocktime.c */
#include <sys/syscall.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/time.h>

int main() {
  struct timespec ts;
  unsigned long long i;

  //750,000,000 calls
  for(i=0; i<750000000; i++) {
    syscall(SYS_clock_gettime,
        CLOCK_REALTIME, &ts);
  }
  return EXIT_SUCCESS;
}
```

Figure A.2.1 Example C Program for get_clocktime system call.

While this design will have to change in future work to improve predictions, we wanted to be as transparent as possible in how we acquired average battery consumption for each system call so far.

Analysis of the Effective Use of Thread-Level Parallelism in Mobile Applications

A preliminary study on iOS and Android devices

Ethan Bogdan
Swarthmore College
ebogdan1

Hongin Yun
Swarthmore College
hyun1

Abstract

Following the model of earlier studies of desktop software, we perform a survey of thread-level parallelism in common mobile applications. In particular, we study the Android and iOS platforms, through a representative sample of 3rd-party software in their respective app stores. Ultimately, we conclude that multiple cores may not be necessary for the majority of mobile experiences, observing that iOS apps tend to have slightly greater average parallelism, with slightly lower variance, than Android apps.

Keywords multi-core, thread-level parallelism, mobile apps

1. Introduction

In chip engineering, there are two broad strategies for increasing computational speed: design processors that can sustain higher clock rates, and design systems that incorporate more processing cores. The former strategy has more or less hit a plateau, as overheating and the limitations of modern cooling systems have come to present an inherent physical barrier. Therefore, in recent years, the majority of time and energy has instead gone into developing more parallel hardware: multiprocessors containing several cores each, and computer systems that contain one or more of these multprocessing chips. However, while this frontier continues to advance, its computational benefits remain limited by the structure of the software that it serves: to take advantage of parallel hardware, developers must practice good parallel design patterns.

We suspect that many systems do not fully utilize their parallel capacities, and, in this paper, we seek to extend the existing body of research in this area into the realm of mobile devices.

2. Background and related work

As a field of computer science, parallel design has a relatively long history, dating back to the early days of servers and networked computers. We draw our inspiration from a couple of more recent studies.

2.1 Parallelism on desktop workstations

In 2010, Flautner et al. studied a range of desktop applications running on Microsoft Windows 7 and Apple's OS X Snow Leopard, analyzing for parallelism. [10] Using the metric of Thread Level Parallelism, or TLP (see section 3.1), they concluded that 2-3 cores were more than sufficient for most applications, and that current desktop applications were not fully utilizing multi-core architectures.

Other studies in a similar vein date back to 2000, when Flautner et al. first investigated the thread-level parallelism and interactive response time of desktop applications. [9] This study was done when multiprocessing was prevalent mostly in servers and had only just begun to enter into desktop machines. While servers were considered to be a natural fit for multiprocessing, due to the parallel nature of serving multiple clients, the benefits of multiprocessing for desktop applications were not obvious. Now, as multiprocessor systems are entering the smartphone market, we believe it is a natural extension of these studies to ask whether the benefits of multiprocessing are fully realized on mobile devices.

2.2 Multi-processing in Android phones

Since its initial release in May 2007, Android and the devices that run Android OS have evolved rapidly. The first commercially available phone to run Android was the HTC Dream, released on October 22, 2008. [3] HTC Dream had a Qualcomm MSM7201A chipset, including an ARM11 application processor, ARM9 modem, and high-performance digital signal processors. [12] In 2010, Google and several handset manufacturers launched a line of smartphones and tablets as their flagship Android devices under the name Nexus. The Nexus One, manufactured by HTC in January 2010, was released with Android 2.1 and had a Qualcomm QSD 8250 with a single core Qualcomm Scorpion CPU [5]. The world's first Android device with a multi-core processor was the LG Optimus 2X, which was equipped with NVIDIA Tegra 2 system-on-a-chip and a 1 GHz dual-core processor. [4]

The latest Samsung Galaxy S4 and Galaxy Note 3, released respectively in March and September 2013, are

equipped with multiple-core processors. All versions of Galaxy S4 and Note 3 are equipped with quad-core processors, and flagship models shipped to certain markets are even equipped with octa-core processors. Both Galaxy S4 and Galaxy Note 3 can run the latest Android 4.3 Jelly Bean. [1] [2]

2.3 Multi-processing in iPhones

Historically, Apple has been very tight-lipped about the internal components of its products – iPhone processors included. However, with a little bit of investigative work, various third parties have determined that the first two iPhone models, released in 2007 and 2008 respectively, both had single-core processors clocked at around 412 MHz. In 2009, with the release of the purportedly fast 3GS model (the "S" stood for "speed"), the iPhone got a boost to 600 MHz. Empirical data shows that subsequent iPhone models have had variable speed processors, ranging from 750 MHz at the low end (of the iPhone 4) to 1.3 GHz at the high end (of the iPhone 5S). Speculation has it that these models use their variable clock rates to conserve energy whenever possible, an important consideration on mobile devices. [8]

It wasn't until the iPhone 4S that Apple first began using dual-core processors, with the introduction of their A5 chip in 2011. They threw in an improved version of this chip, along with a three-core graphical processor, in the iPhone 5.

Interestingly, it had been a year earlier, in 2010, that Apple released the first version of iOS to support multi-tasking, iOS 4. This release was further notable for including Grand Central Dispatch, a new technology that offered developers a simple, high-level interface for efficient thread management. The operating system first shipped on the iPhone 4, which, with its single-core A4 processor, could not support genuine parallelism. Nevertheless, the A4 was sufficiently fast to achieve an illusion of concurrency, and the new operating system and multi-threading technology helped pave the way for the advent of the 4S the following year.

3. Methodology

In the interests of consistency, we have tried to model our methodology on the aforementioned work by Flautner et al. In this section, we will summarize that approach, then go on to discuss the particularities of working with Android and iOS systems. It is important to note that we carried out all of our research using actual, physical phones. This hardware allowed a far higher degree of accuracy and transparency in our data collection than any kind of emulation would have offered.

3.1 Metrics

There are several different common metrics for quantifying parallelism in computer systems.[9][10] One simple and intuitive metric is Machine Utilization, which is a measure of the percentage of total processing resources that gets used during execution. The formula for Machine Utilization is shown in Equation 1:

$$\text{Machine Utilization} = \frac{\sum_{i=1}^{n} c_i i}{n} \tag{1}$$

In this equation, n is the number of thread contexts in the subject machine, and c_i is the fraction of time that $i = 0, ..., n$ number of threads were executed concurrently. If all processors in the machine were fully utilized during the execution of the benchmark, Machine Utilization would be 1. This intuitive metric is, however, not suitiable for the type of study we conducted. Applications on mobile devices tend to incur a significant amount of idle time, when no threads are being executed in any of the processors, due to a high degree of user interactivity and I/O activity.

We therefore decided to use Thread Level Parallelism (TLP), the same metric that Flautner et al. used in their pioneering research. TLP is a variation of Machine Utilization that factors out idle time. The formula for TLP is given in Equation 2:

$$\text{TLP} = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \tag{2}$$

As in Equation 1, n is the number of thread contexts in the subject machine, and c_i is the fraction of time that i threads were executed concurrently. TLP output values will fall between 1 and n.

One last caveat we had to consider was how exactly to measure c_0, the idle time of the system. While this measurement might be straightforward on a relatively simple operating system, both Android and iOS are constantly running a host of background processes. Some of these processes are indispensible to the active app, while others may serve entirely unrelated functions of the system; (for example, they might monitor cellular connectivity). Our goal was to measure each app in as much isolation as possible, but, since we had no sure way of determining which background processes were a part of the app and which weren't, we ran all of our TLP calculations twice. The first time, we assumed that all background processes constituted "idle" time for the foreground app, while the second time we assumed that they were "active." The reality is most likely somewhere in the middle (see section 5 for further discussion).

3.2 Benchmarks

We conducted two separate sets of experiments. The first was intended to encompass a broad and representative cross-section of applications currently available on the Android Market and iTunes app stores. Our sampling policy involved selecting three applications from each of seven popular categories:

1. Business
2. Entertainment
3. Games (Action)
4. Games (Puzzle)
5. Media
6. News
7. Social Networking

To ensure that these applications accounted for a substantial amount of user experience, we pseudorandomly selected them out of the 50 most popular free apps for each category.[1]

For our second set, we handpicked 10 apps that had cross-platform success on both Android and iOS. We found two such apps in each of five popular categories:

1. Entertainment
2. Games
3. Media
4. Productivity
5. Social Networking

The goal here was to conduct a more focused study of how each mobile operating system handles parallelism differently, by controlling for the particular software being run.

We collected data on each of these benchmarks in a three-pass process. The first pass involved exploring basic functionality without our debugging software running. This pass enabled us to gain familiarity with each app; observe its baseline behavior; and take care of setting up any accounts, granting any permissions, completing any tutorials, etc. that would be required on an initial run. During our second and third passes, we recorded data while manually engaging the central features of each app, as we had previously determined them. The purpose of recording in two passes was to help control for noise associated with background processes that we did not have control over. As much as possible, we tried to reproduce the behavior of the second pass during the third pass, based on careful notes of the input we had provided.

On Android, each of the second and third passes spanned a duration of 90 seconds, while they spanned 45 seconds on iOS (see section 3.4.1). This duration always included the opening and start-up time for each app.

3.3 Android setup

3.3.1 Systrace

Systrace helps analyze the performance of an application by capturing that application and other Android system processes, then representing them in a graphical format.[6] The tool comes with the Android Software Development Kit (SDK), available for free at the official Android developer website. In order for an application's activity to be traced, the application must be run on a physical device connected to a developing system via USB. Systrace, which runs as a Python script on the developing machine, then establishes a debugging connection via the Android Debugging Bridge (ADB). Systrace calls ATrace, a native Android binary, via ADB, and then ATrace in turn uses FTrace to capture kernel events. FTrace is a Linux kernel tool for tracing function execution in the Linux kernel. FTrace operates through instrumenting kernel functions; when the kernel is configured to support function tracing, the compiler adds code to the prologue of each function.[7] This routine does cause some overhead to the application that is being traced, but quantifying the exact amount of overhead is outside the scope of this paper.

Systrace outputs combined data from the Android kernel and generates an HTML report that gives an overview of every activity that was processed on the device for a given period of time. The output trace file shows a detailed overview of CPU activity, including process name, start time, process and thread id, and the CPU on which the process was executed. We built a Python script of our own to parse the string output data and compute the duration and TLP for all cores in the subject system.

The version of Systrace that we used came included in the Android SDK 4.2 (API 17).

3.3.2 Hardware

Our Android device was a Samsung Galaxy S3 I747, which has a Qualcomm MSM8960 Snapdragon chipset with 1.5 GHz Advanced Dual-core. The operating system on the device was upgraded to version 4.1.2.

3.4 iOS setup

3.4.1 Instruments

Apple restricts most low-level access to system information on iOS, but it does provide Instruments – a free, first-party debugging package, bundled with every download of Apple's Xcode IDE. Instruments incorporates a wide variety of tools for measuring anything from backlight brightness and battery usage to zombie processes and memory leaks. These tools, which permit some degree of customization, are essentially a front end to DTrace, a common dynamic instrumentation program for Unix-based systems. DTrace itself relies on making modifications to the system kernel, but Apple does not allow third-party developers that privilege directly.

Through Instruments, we ran the Time Profiler tool, which claims to perform "low-overhead time-based sampling of processes running on the system's CPUs." This tool does not have a visible impact on the operation of most apps, but it does seem to incur a fair amount of overheard on the computer system that runs it. Although our initial goal had been to record over 90-second time windows, trial runs indicated that this duration would consistently cause Time Profiler to hang indefinitely, requiring a force quit. We therefore chose to record over a less taxing 45-second range, which

[1] We used the Python `random` library to generate pseudorandom values. We had to reject a couple of apps on the basis of requiring preexisting accounts or coroporate affiliations.

was still unreliable, but resulted in crashes only around 50% of the time. [2]

We used the latest version of Instruments at the time of our research, v5.0.1, running on Mac OS 10.9.

3.4.2 System preparation

In order to further reduce noise in our data and enhance the isolation of the foreground app, we deactivated several of the automatic features of iOS 7. Among these features were Background App Refresh, (which schedules apps to run time-limited tasks in anticipation of their next use) and the parallax effect, (which acceses accelerometer data to re-render UI elements at high frequency). Both of these features are easily controlled via the Settings menu. Finally, we made sure to open the iOS 7 app switcher and kill all apps before the start of each data collection pass.

3.4.3 Hardware

Our test device was a 16 GB iPhone 4S with a dual-core A5 processor running at around 800 MHz. At the time of data collection, it had just over 1 GB of free space and was running iOS 7.0.4. We connected it via USB 2.0 to a 2011 iMac with a 3.1 GHz Intel Core i5 processor and 8 GB of RAM.

4. Experimental results and analysis

In total, we collected data on 52 distinct apps – 21 for Android, 21 for iOS, and 10 that ran on both platforms. The arithmetic mean of all TLP values we calculated was around 1.28, indicating a low, yet non-negligible degree of parallelism among mobile apps. Generally speaking, TLP values were significantly higher when calculated with background processes counted as a part of the active process than when calculated with background processes as idle. Likewise, TLP values for iOS tended to be higher than those for Android, on the order of 1.33 to 1.22 for TLP w/background and 1.15 to 1.00 for TLP w/o background.

At the extreme ends of our TLP w/background ranges, we had values as low as 1.02 (for Android) and as high as 1.70 (also for Android). In fact, the variance for our Android data overall (0.028) was significantly higher than the variance for our iOS data (0.008).

Beyond these broad comparisons, there were no obvious patterns that emerged in our data. Category or genre of app does not appear to be a strong determinant of TLP, although it is true that, in our direct comparison of 10 apps, both platforms had maximal TLP values for Entertainment and Social Networking apps and minimal TLP values for Games.

For a complete listing of our data, please consult the tables and figures on pages 6-9 of this paper.

[2] We confirmed this crash rate on multiple systems, suggesting that it was not an error in our setup. It remains unclear why Instruments could not handle the full 90 seconds, even despite a large buffer size.

5. Discussion

Trying to suss out the primary culprit behind our low TLP values is a difficult business, since actual parallelism is a product of hardware, application, and operating system combined.

On the hardware side, even a phone equipped with a multi-core processor may redirect its most parallel computations to a designated GPU, for which we have no source of data. (There's a decent chance that this divison of labor is precisely why our Games category had some of the lowest TLP on the CPU.) The hardware may also shut down one of its cores altogether to conserve energy, preempting any potential for parallelism. (Although, given that our devices were receiving continuous USB power throughout data collection, this possibility seems unlikely.)

On the application side, even an app with many threads may be designed to offload the bulk of its processing to a remote server, then retrieve the results in a less parallel fashion. (On iOS, the Pho.to Lab app functioned exactly like this.)

And, lastly, the operating system needs to do the actual scheduling of those threads, and it is anything but predictable how the OS will balance the work of a single application alongside other duties of the system.

Amidst all this uncertainty, we can minimally conclude that counting background processes as non-idle work, associated with the foreground app, is the better metric to be using for TLP. On the Android side, parallelism was practically nonexistent under the other metric, which runs contrary to our intuition that at least a few apps should have good parallel design. Meanwhile, on the iOS side, the primary two background processes we observed were `SpringBoard` and `backboardd`, both of which handle crucial, UI-related services for all apps, (such as processing taps and gestures on the multi-touch display). The activity of these processes is thus inherently tied to the activity of the foreground app, and we would be remiss to count it as idle.

Further discussion of issues specific to one operating system or the other follows below:

5.1 Android

The aforementioned high TLP value of 1.70 was a definite anomaly, produced by an application called Drippler. Drippler is a rather simple news and magazine app that provides daily tips and updates about the mobile devices with which the user accesses it. We did not have access to the source code, but, upon examining the trace data of the app, we found out that it uses the thread pool function, which is a built-in Java technology for generating and managing threads. We suspect that thread pool was what enabled and facilitated Drippler's strongly parallel behavior.

5.2 iOS

If there is one obvious reason why iOS apps would have consistently higher TLP values than Android apps, it's Grand Central Dispatch (GCD). Managing pthreads and mutex locks at a low level can be intimidating for many developers, resulting in a high barrier to entry for parallel design. However, using GCD is a very straightforward process, for which there exists a wealth of official documentation and accessible tutorials. As a testament to the ubiquitousness of this technology, it is an industry best practice for iOS developers to assign all heavy computation to background threads via GCD, in order to avoid UI hangs. Thus it is logical to assume both that GCD would increase the amount of parallelism in an average iOS app and that it would yield fairly consistent TLP values across the board, given that it is the single go-to strategy for managing threads.

6. Future directions

For this paper, our goal was to study apps in isolation as much as possible. This would allow us to determine whether or not mobile developers were building good parallel structure into their own apps, on an individual basis. However, these data alone cannot answer the broader question of whether multiprocessors are right for smartphones. For instance, it could be the case that typical usage of one of these devices – which involves listening to music, checking e-mail, tracking geolocation, and downloading software updates all at once – actually yields far higher TLP, close to 2.00. In this case, multiprocessors would be an invaluable component of mobile systems, even if individual app developers don't know how to make good use of them. Thus, future work might include recording data based on other usage patterns.

Another obvious realm to explore would be the third major mobile operating system: Windows Mobile. While Windows-based phones currently hold less than 5% market share, they are on the rise and already twice as prevalent as the next closest runner-up (Blackberry).[11]

Finally, on all of these platforms, it would be worth collecting additional data, both to increase the rigor of our existing conclusions and to look into related questions. These questions might include: how consistent TLP results are across many runs of the same app; what kinds of threading behavior different apps exhibit; and how much parallelism is handled by the GPU.

7. Conclusion

For the vast majority of mobile applications, having access to two cores on a fast, modern multiprocessor seems to be overkill. Either the current demand for software does not require it, or developers do not currently know how to meet a demand that does. Nonetheless, there is reason to believe that improved tools for handling multi-threading, such as Java thread pools and Apple's Grand Central Dispatch, may already be helping developers bridge the gap between physical capacity and realized potential. We believe that there remains room for further progress on these sorts of high-level technologies.

Acknowledgments

We would like to thank Ben Ylvisaker for helping us direct our line of inquiry and for collaborating with us as we continue this research.

References

[1] Galaxy Note 3 Specification. http://www.samsung.com/us/mobile/cell-phones/SM-N900AZKEATT-specs.

[2] Galaxy S4 specification. http://www.samsung.com/us/mobile/cell-phones/SGH-I337ZRAATT-specs.

[3] T-mobile press release. http://www.t-mobile.com/company/PressReleases_Article.aspx?assetName=Prs_Prs_20080923&title=T-Mobile%20Unveils%20the%20T-Mobile%20G1%20%E2%80%93%20the%20First%20Phone%20Powered%20by%20Android.

[4] LG press release. http://www.lg.com/global/press-release/article/lg-launches-world-first-and-fastest-dual-core-smartphone.jsp.

[5] "Nexus One Owner's Guide," Google, Tech. Rep., 2010.

[6] Systrace. http://developer.android.com/tools/help/systrace.html.

[7] T. Bird, "Measuring function duration with ftrace," 2009.

[8] EveryiPhone.com, "What processor or processors do the iPhone models use?" last accessed 12/2013. [Online]. Available: http://www.everymac.com/systems/apple/iphone/iphone-faq/iphone-processor-types.html

[9] K. Flautner, R. Uhlig, S. Reinhardt, and T.Mudge, "Thread-level parallelism of desktop applications," *Workshop on Multi-threaded Execution, Architecture, and Compilation*, 2000.

[10] K. Flautner, G. Blake, R. G. Dreslinski, and T.Mudge, "Evolution of thread-level parallelism in desktop applications," *SIGARCH Computer Architecture News*, vol. 38, pp. 302–313, 2010.

[11] D. Kerr. (2013, November) Android dominates 81 percent of world smartphone market. [Online]. Available: http://news.cnet.com/8301-1035_3-57612057-94/android-dominates-81-percent-of-world-smartphone-market/

[12] L. Zhang, B. Tiwana, Z. Qian, and Z. Wang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," *CODES/ISSS '10 Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2010.

	Android			iOS		
Category	**App**	**TLP (w/back.)**	**TLP (w/o back.)**	**App**	**TLP (w/back.)**	**TLP (w/o back.)**
Business	Backpage	1.28	1.00	Fiverr	1.43	1.31
	Olive Office	1.08	1.00	QuickVoice	1.21	1.06
	Vault-Hide	1.38	1.00	Photo Collage Free	1.18	1.09
	Mean:	**1.25**	**1.00**	**Mean:**	**1.27**	**1.16**
Games - Action	Battle of Zombies	1.04	1.00	Temple Run	1.23	1.07
	Jetpack Joyride	1.03	1.00	Dark District	1.30	1.18
	Skee-Ball	1.21	1.00	Real Steel World Robot Boxing	1.39	1.27
	Mean:	**1.09**	**1.00**	**Mean:**	**1.30**	**1.17**
Games - Puzzle	Fruits & Berries	1.02	1.00	Disco Bees	1.35	1.16
	Heads Up Charades	1.09	1.00	Can You Escape?	1.25	1.06
	Where's My Water 2	1.22	1.02	Bejeweled Blitz	1.26	1.09
	Mean:	**1.11**	**1.01**	**Mean:**	**1.29**	**1.10**
Entertainment	Best Vines	1.19	1.00	TwitchTV	1.31	1.25
	Bitstrips	1.24	1.00	Bell for Christmas	1.15	1.04
	Watch ABC	1.15	1.02	CamWow	1.33	1.08
	Mean:	**1.19**	**1.01**	**Mean:**	**1.26**	**1.12**
Media	BS Player	1.12	1.00	Vimeo	1.33	1.13
	Torrent Video Player	1.05	1.00	InstaSize	1.29	1.14
	Photo Locker	1.39	1.00	Pho.to Lab	1.40	1.16
	Mean:	**1.19**	**1.00**	**Mean:**	**1.34**	**1.14**
News	Drippler	1.70	1.00	BBC News	1.44	1.23
	Lotto Results	1.21	1.00	iCitizen	1.36	1.22
	Yahoo	1.17	1.00	AARP	1.45	1.27
	Mean:	**1.36**	**1.00**	**Mean:**	**1.42**	**1.24**
Social Networking	Emojidom Smileys	1.47	1.00	LinkedIn	1.43	1.13
	Foursquare	1.36	1.00	Kik Messenger	1.38	1.13
	Text Free	1.17	1.00	OKCupid	1.41	1.15
	Mean:	**1.33**	**1.00**	**Mean:**	**1.41**	**1.14**
	Overall mean:	**1.22**	**1.00**	**Overall mean:**	**1.33**	**1.15**
	Variance:	**0.028**	**0.000**	**Variance:**	**0.008**	**0.006**

Table 1: TLP data by category and operating system
(w/back. = counting background as part of the active process)
(w/o back. = counting background as idle)

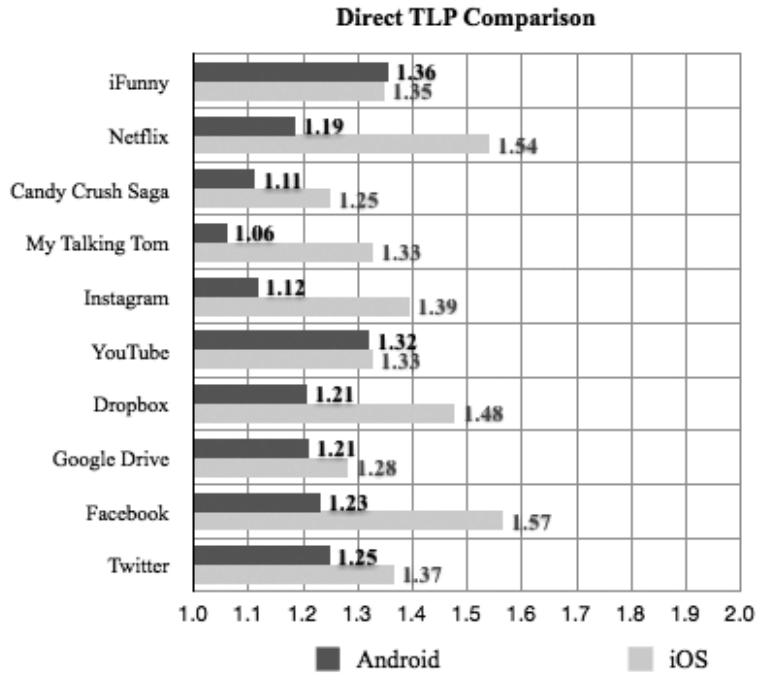

Figure 1

Category	App	Android TLP	iOS TLP
Entertainment	iFunny	1.36	1.35
	Netflix	1.19	1.54
	Mean:	**1.27**	**1.44**
Games	Candy Crush Saga	1.11	1.25
	My Talking Tom	1.06	1.33
	Mean:	**1.09**	**1.29**
Media	Instagram	1.12	1.39
	YouTube	1.32	1.33
	Mean:	**1.22**	**1.36**
Productivity	Dropbox	1.21	1.48
	Google Drive	1.21	1.28
	Mean:	**1.21**	**1.38**
Social Networking	Facebook	1.23	1.57
	Twitter	1.25	1.37
	Mean:	**1.24**	**1.47**
	Overall mean:	**1.20**	**1.39**
	Variance:	**0.008**	**0.011**

Table 2: Direct TLP comparison data
(calculated with background as part of the active process)

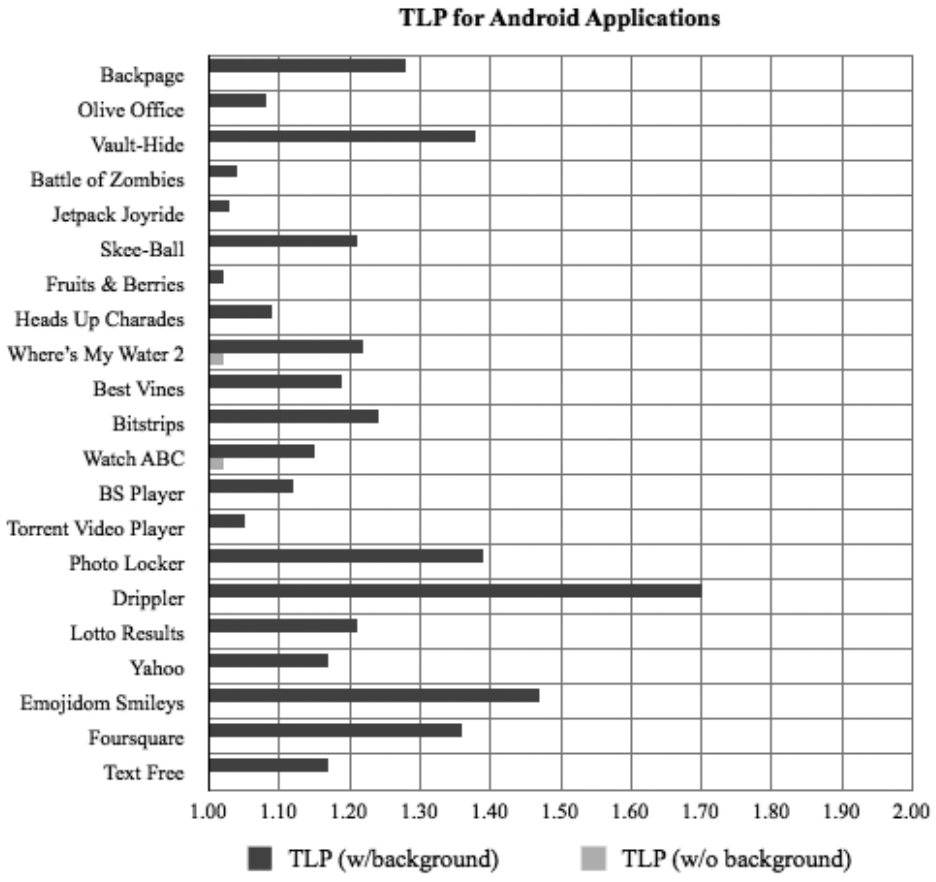
TLP for Android Applications
Backpage
Olive Office
Vault-Hide
Battle of Zombies
Jetpack Joyride
Skee-Ball
Fruits & Berries
Heads Up Charades
Where's My Water 2
Best Vines
Bitstrips
Watch ABC
BS Player
Torrent Video Player
Photo Locker
Drippler
Lotto Results
Yahoo
Emojidom Smileys
Foursquare
Text Free
1.00
1.10
1.20
1.30
1.40
1.50
1.60
1.70
1.80
1.90
2.00
TLP (w/background)
TLP (w/o background)

Figure 2

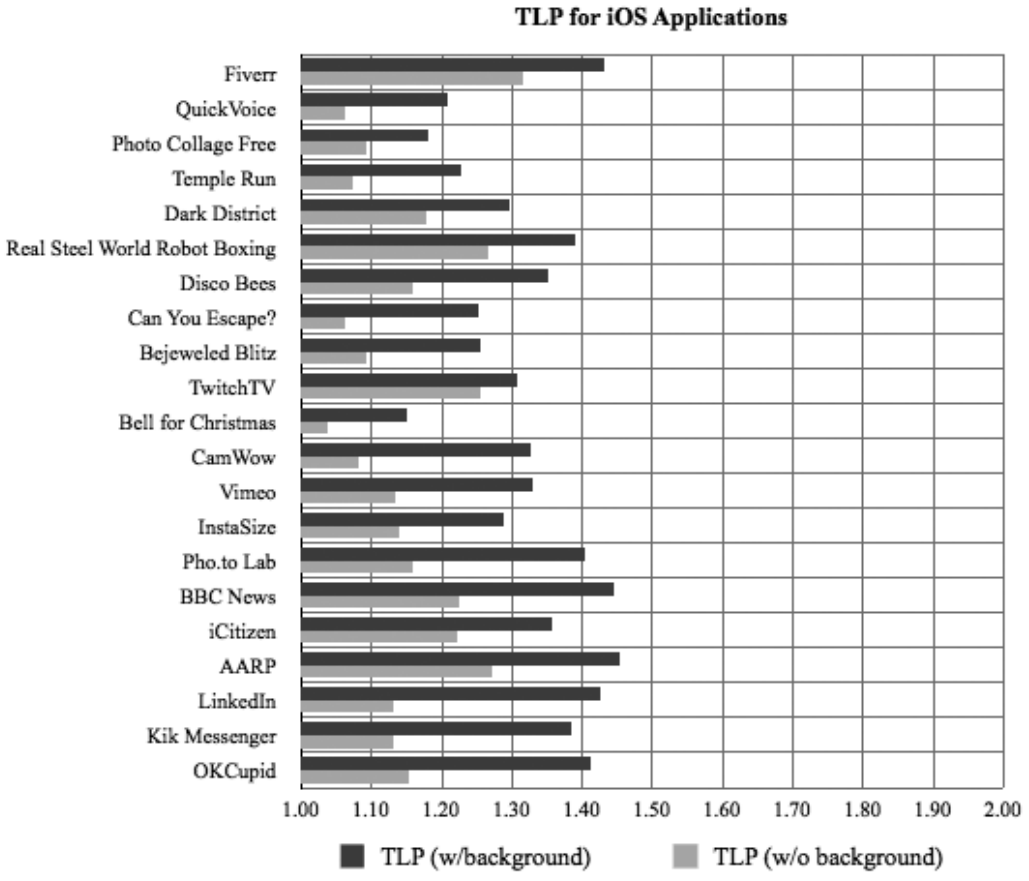
TLP for iOS Applications
Fiverr
QuickVoice
Photo Collage Free
Temple Run
Dark District
Real Steel World Robot Boxing
Disco Bees
Can You Escape?
Bejeweled Blitz
TwitchTV
Bell for Christmas
CamWow
Vimeo
InstaSize
Pho.to Lab
BBC News
iCitizen
AARP
LinkedIn
Kik Messenger
OKCupid
1.00
1.10
1.20
1.30
1.40
1.50
1.60
1.70
1.80
1.90
2.00
TLP (w/background)
TLP (w/o background)

Figure 3

Usejax: Evaluating Usability of AJAX Websites through Crawling

Peter Ballen
Swarthmore College

Kevin Roberts
Swarthmore College

Abstract

With more and more business being built online, there is a need for fast, accurate, and automated usability testing tools. However, as the web expands and evolves, our tools must also evolve. Websites built using AJAX provide new problems for traditional task based evaluation methods because a single url could correspond with multiple states.

We build the tool UseJax to discover whether an automated usability tool can work with AJAX webpages and still approximate human behavior. We implemented UseJax as a plugin for CrawlJax, a webcrawler designed to index AJAX webpages. UseJax has two phases, a crawl phase where a graph of the website is constructed, and an analysis phase where UseJax applies a keyword based heuristic to simulate human behavior traversing the website.

Our tests of UseJax confirm that we are able to find states without a unique url. We were also able to use site search. However, UseJax was less capable at solving tasks than humans. If UseJax could solve a task, humans could almost always solve the same task. But UseJax had difficulty solving tasks that humans solved with ease. Furthermore, using CrawlJax as a base for UseJax limits its effectiveness as a tool: UseJax crawls the entire webpage as opposed to a directed crawl. Large websites with large branching factors will lead to long crawl times and as a result long analysis times.

1 Introduction

The goal of web usability testing is to determine how easy a site is to navigate. A significant portion of usability testing is task based. Traditionally, a human subject is given a series of tasks to accomplish on a website. Sample tasks might involve purchasing an item or finding information about a noteworthy individual. The subject accomplishes the task while a researcher observes his or her actions. The researcher gathers both qualitative and quantitative data. Was the website frustrating to use? Did any of the buttons not behave as the user expected? How long did the user take to complete the task? How many links did he click?

In this day and age, even the smallest businesses are expected to have some sort of web presence, and if these websites are not user friendly it may cost them customers or business opportunities. While they may be able to create a website, many small companies may not have the resources to hire private user experience firms or the knowledge to conduct usability testing. Using human subjects to test a website is a slow and costly process.

Recently, a number of tools have been developed to automate some or all of this process. To our knowledge, no tool exists that attempts to model frustration or similar qualitative features. The majority of automated testing has focused on gathering quantitative data. In particular, the number of clicks a user needs to complete a given task is a commonly discussed metric. If a large number of clicks are required to complete an important task, the website may have serious design flaws.

The number of links required to solve a task is not the only assessment of usability, if usability was that simple then a simple web crawler could preform usability testing. A second important factor is whether following those links is intuitive. If the clicks are easy for the user to find and interact with, then the user may not mind if solving a task requires a large number of clicks. Conversely, a website that could complete any task with a single click would have a homepage so cluttered that users would find it hard to find anything that they wanted. Our tool tries to approximate human behavior, following links that humans are likely to click on.

Other tools exist for discovering usability issues in websites. However, previous techniques have focused on entirely page-based websites and neglected websites built using AJAX. More and more websites are using AJAX in their designs. In this paper, we introduce Use-

Jax, a tool developed to automate usability testing on AJAX webpages.

In addition to finding information this way, we use a site's search functionality when there is no clearly visible link to better approximate human behavior. One drawback to this approach is that the web crawling application time depends on the size of the site. Large websites may take too long to crawl. However, smaller companies with smaller websites are most in need of a cheap and quick usability testing tool. UseJax excells at evaluating these smaller websites.

2 Background

UseJax looks to bring together usability testing, AJAX, and usability automation, three topics about which there are varying degrees of documentation. What we wanted to focus on are some of the ways these pieces interact and how they are going to be used by UseJax.

2.1 Usability Testing

There are a lot of pieces of information about how to make a website with a positive user experience. Some of the most important qualities are usability, accessiblity, and findability. Usability refers to a general ease of use for the website. Accessibility means that all users should be able to access the page's content, regardless of disability or computer skill. Findability simply means that the content needs to be easily found. We focused on the issue of findability because it is the most readily automatable.

There were also some less official usability rules we wanted to take into account. One of the most commonly repeated is the three click rule. The three click rule states that websites should be designed so that no content is more than three clicks away. However, there is also evidence showing that user satisfaction is not actually strongly tied to number of [1]. In consideration of this rule, we felt we could limit the depth our tests of user approximation. We decided that UseJax by default follows the three click rule, and gives up trying to solve any task that requires more than three clicks.

The majority of automated usability tools we found were not actually fully automated, in that they were not using just a program and offline inputs. Instead, most tools were added into the test site while it was online so that the tool could collect live data about how users were interacting with the site. Others looked at other potential issues a user might run into, such as the readability of the content of webpage or the load time. These were definitely helpful tools, with readability going into the idea of web accessibility and load time factoring into how usable people feel a site is. However, we didn't feel these tools were truly automating the process of usability testing.

One tool which did look at findability issues through an actually automated system was the Bloodhound Project[2]. It was able to navigate webpages using Infoscent Simulations to approximate how users navigate along a website using "proximal cues" such as font size and color and other information they already know or learned about the goal. This technique resulted in moderate to strong correlations with actual website users. However, these results omit any users which used built-in website search functions or did not leave the homepage, ignoring information which could be found through state changes but without changing the URL. We did not expect our system to reach the level of Bloodhound for approximating humans, but we did want to be able to take advantage of AJAX and search where their system could not.

Site search is implemented in most current websites and we felt a usability tool should be able to make at least some use of it. However, how much do users use site search? Research suggests that less than one third of people are finding the page they need via site search immediatly; most people start by browsing the links on the site [3]. In fact, during our own usability tests, we found that humans never took advantage of site search.

2.2 AJAX and Usability

In traditional page-based sites, the only way to change the user's perception is to load a new page with a new url. This leads to some awkward behavior: changing one line of text requires loading an entire new page. AJAX (Asynchronous Javascript and XML) is a web-developing technique that allows the programmer to change a page's content without loading a new page. AJAX can dynamically insert elements, such as messages or links, into the Document Object Model (DOM) of a webpage.

For the purposes of this paper we are defining an AJAX state change as a change that modifies the DOM and user perception of the page, but does not have a new url. AJAX poses new challenges for usability [4]. The main challenge is that AJAX redefines what makes up a "state." In a page-based system, each state is represented with a unique url. In an AJAX-based system, a single url could be shared by multiple states. This interferes with some common browser functions, such as the back button or bookmarks, because it is unclear which state a user wants to go back to or save.

The notion of unclear states also affects automated usability tests. For a task-based usability test, the url of the target state is not sufficient to distinguish that state. To set up the test, unique elements from the DOM of the tar-

get state might be taken out and pointed to. We also have to be able to distinguish when we are in these unique states and what paths were taken to reach them.

2.3 CrawlJax

The tool we are building UseJax off of is CrawlJax. CrawlJax is a tool designed to dynamically look through AJAX webpages [5]. By looking at the Document Object Model (DOM) of a webpage, it can look for user interactions which could trigger a change in the DOM tree. If a new state is different enough, it is added to a state-flow graph. This API has also been used for determining accessibility of AJAX applications [6] as well as for clustering websites based on certain features [7].

3 UseJax

UseJax is our usability testing tool. When given a simple task, UseJax will simulate how a human would attempt to solve the task and record the number of clicks needed.

We choose to focus on information retrival tasks as these are the easiest to automate. It is possible to automate other tasks, such as online shopping. However, many more complicated tasks can be reduced to an information retrival task. For example, an online shopping task, "Purchase a new Dell laptop" can be rephrased as "Find the price of a new Dell laptop" with the hope that a user who found information about the laptop could then see the "Buy Now" button.

UseJax tasks are very simple. A start url and goal url are manually inputted into UseJax. The task is solved by finding a series of clicks that leads from the start url to the goal url. On AJAX webpages, it is possible that different AJAX states have the same url. Thus, UseJax can be configured to add conditions to the goal. For example, a sample goal might be "find the url www.sample.com/target and make sure the message 'You found the goal' appears in the DOM tree". Thus, UseJax can distinguish between multiple AJAX states with the same url.

UseJax uses a two-phase process to solve tasks: the crawl phase and the analysis phase. The crawl phase builds a graph that represents the website, and the analysis phase simulates human behavior on this graph.

3.1 Crawl Phase

The first step to completing a task is the crawl phase. During the crawl phase, UseJax takes advantage of the Crawljax interface to crawl a website.

UseJax begins its crawl by loading the start url and identifying all eventables on the page. A eventable is any object that triggers a dom change. The most common

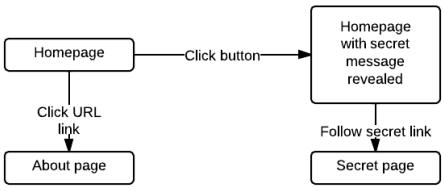

Figure 2: Sample State Flow Graph

eventable is a url link that loads a new page. Other types of eventables are buttons and hover text. UseJax then triggers each eventable in turn, looks at the new DOM, and recursively repeats this process.

In order to take advantage of site search, UseJax must be given the search term prior to the crawl phase. Once in the crawl phase, it will fill any input field with the given search term. When the eventable connected to that input field is eventually interacted with, the next state will be the search results of that term. The CrawlJax interface allows only one search term per input field type, and this must be inputted pre-crawl. This means UseJax cannot conduct multiple searches in a single crawl phase. This means we cannot evaluate search on more than one target state during a single run. It also means UseJax cannot handle more complicated input field systems like one might find on a hotel booking website.

As proof of concept, we constructed a simple example. In this example, the start url has two eventables: a link to the about page and a button. Clicking on the button triggers a DOM change, a secret message and link appear. The url does not change: this button does not load a new page, but rather modifies the DOM of the current page. Figure 1 illustrates this change.

The end result of the Crawl Phase is a directed State Flow Graph. AJAX states are vertices in the graph. Every vertex represents a unique DOM state. Note that two vertices could have the same URL, but different DOM states. One AJAX state is connected to the other if it is possible to obtain the second state from the first by triggering an eventable. The state flow graph of the previously described example is illustrated in Figure 2

This simple examples already exposes the strength of UseJax and the weakness of the page-based approach. Observe the somewhat convoluted path required to find the secret page from the start page: load the start page, identify that the button triggers a DOM change, click the button, look through at the modified DOM, realize that a new link has appeared, and finally click on the link. A conventional page based approach could never access the secret page, whereas UseJax finds it with ease.

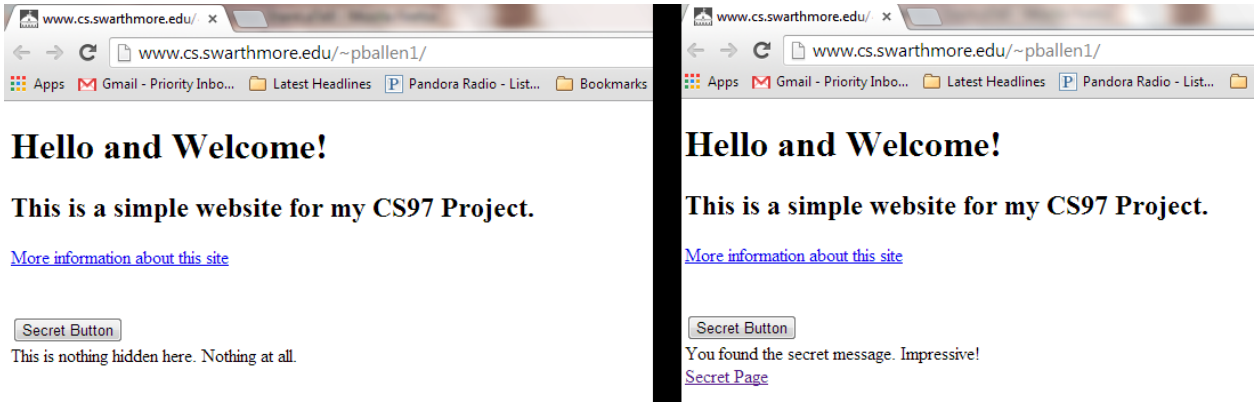

Figure 1: On left, homepage. On right, homepage after DOM change has been triggered.

3.2 Analysis Phase

Once the crawl phase is complete, the analysis phase begins. Using the state flow graph, we attempt to find a path from the start state to the target state. We use a simple greedy search algorithm, implementing a number of heuristics designed to approximate human behavior. We constructed four heuristics to use when searching for a path from the start state to the target state.

The first heuristic we constructed was keyword based. Given a sorted list of keywords, the search chooses clickables based on the keyword list, with higher ranked words getting priority over lower ranking words. For example, if we wanted to find information about buying a laptop, we might use [”Inspirion 15”, ”Laptop”] as our keyword list. The purpose of this heuristic was to be a very simple yet robust way of navigating a website, especially when the user has no prior knowledge of the website or the goods it is offering.

The second heuristic is similar to the first heuristic, but is much more specific and therefore a more important assessor of usability. It used only the first keyword of the first heuristic, usually the title of the target webpage or a slightly shortened form, and then looked for it only within the title or header tags in the HTML. Originally this heuristic was meant to capture things which were of larger sizes than the things around them, but Crawljax does not provide access to the CSS of the states it is saving after it has crawled.

The third heuristic was programmed to take advantage of site search. Almost every web page these days has its own internal search, often powered by Google, and is often a useful way to find out of the way pages. However, human beings use search as a first resort less than 30 percent of the time. Even when users are unable to find the page their are looking for, they still not the most likely to use the search. However, we believe that it is an important feature of the site that the search be well implemented enough that a quick scan of the results will provide where to go. Therefore, after using the search the search this heuristic then defaulted to heuristic 1 behavior, clicking all of the results which have matching keywords.

We also combined our heuristics to create a fourth path creating method we thought would best approximate users. It first attempts to use heuristic 2, then heuristic 1, then heuristic 3 for each of its eventables.

We also created an optimal heuristic that assumes perfect knowledge of the website and always finds the optimal path. This is not a realistic simulation of human behavior, but it can be helpful to know the ’optimal’ path to solve a task.

The advantage of splitting UseJax into two seperate phases is that multiple Analysis phases can be run on a single state flow graph. It is very simple to implement new heuristics and have them run alongside of, or instead of, the four implemented heuristics.

4 Evaluation

We tested both the effectiveness of the Crawl phase and the effectiveness of the Analysis phase.

4.1 Crawl Evaluation

To test the effectiveness of the crawl phase, we ran UseJax on four sites. Site1 is the proof of concept page seen in Figure 1. Site2 is the Swarthmore Computer Science department homepage [8]. It is entirely page based and uses no AJAX elements. Site3 is the Swarthmore DASH [9]. It only has a few unique urls, but relies heavily on AJAX. Site4 is the Swarthmore College homepage [10]. It uses AJAX and is substantially larger than the other three sites. In order to speed up the crawl, we implemented a depth limit for each site. Webpages that are too

	Site1	Site2	Site3	Site4
Crawl Successful	Yes	Yes	Yes	No, Aborted
Crawl Time	4s	6 min, 31 s	18 min, 25 s	2 hr
Depth Limit	2	3	1	2
Number of States	4	109	22	1013
URLs Visited	3	109	5	617
Avg DOM Length	0.488 kB	11.747 kB	76.482 kB	60.47 kB
Largest Branching Factor	2	38	21	50

Table 1: Crawl Evaluation

far away from the homepage will not be crawled. We summarize the results of the crawls in Table 1

UseJax is capable of successfully crawling websites that use AJAX and websites that use no AJAX. It is interesting to note that crawling a page-based website is substantially faster than crawling a AJAX-heavy site. Site2, a no-AJAX site, took 18 minutes to crawl. Site3, an AJAX-heavy site, took 2 hours to crawl.

This is a result of the complexity of the DOM. Sites that use AJAX typically have much longer DOMs than sites that use the page-based approach. Site2 had an average DOM length of 11.7kB, whereas Site3 had an average DOM length of 76.4kB. As part of the crawl phase, UseJax need to search through the entire DOM to identify eventables. Thus, a longer average DOM length contributes to a longer Crawl phase.

While UseJax successfully crawled three of the test sites, we terminated the fourth crawl after two hours passed. Site4 has a much larger number of states, and as a result requires much longer to crawl. UseJax has difficulty with large sites. We discuss some implications of this result in Section 5.

4.2 Analysis Evaluation

One goal of UseJax, and automated usability tools in general, is to approximate human behavior. To test how well UseJax succeeds in this task, we ran a series of usability tests. A small sample of humans (5-10 college students) were given a start url and information retrival task. They were told to load the given url and complete the task. The tasks were intentionally chosen to be particularly easy for a human to solve.

For each task, we give the start page, the instruction as given to the human subjects, the most common human solution, and the solution UseJax found.

Task 1
Start Page: Swarthmore CS Department Homepage
Task: "Find information about the recent ACM competition"
Human Solution: Click on ACM Results link in Current CS News section of homepage (1 click)
UseJax Solution: Same as human solution

Task 2
Start Page: Swarthmore CS Department Homepage
Task: "Find information about the CS21 Course"
Human Solution: Click on Courses, then click on CS21 (2 clicks)
UseJax Solution: Same as human solution

Task 3
Start Page: Swarthmore CS Department Homepage
Task: "Find information about Lisa Meeden"
Human Solution: Click on People, then click on Lisa Meeden (2 clicks)
UseJax Solution: Failed

UseJax does not implement natural language processing. When looking for links, UseJax does not consider synonyms or categorizations. While every human was able to make the connection that Lisa Meeden is a person, UseJax could not make the connection. One workaround fix is to add the word 'people' to the keyword list. However, this reveals one of the weaknesses of UseJax; the user may be required to set up the keyword list to help guide UseJax to the proper destination.

Task 4
Start Page: Swarthmore College Homepage
Task: "Find information about McCabe library"
Human Solution: Click on Libraries, then click on McCabe (2 clicks)
UseJax Solution: Failed

Our other tasks on the Swarthmore College Homepage were similarly unsuccessful.

As we mentioned in the previous section, UseJax has difficulty with large sites. The McCabe site was never added to the state flow graph, so there was absolutely no chance for UseJax to find a path from the start state to the McCabe page. However, we feel that if UseJax had been allowed to run longer, it would have eventually inserted the McCabe page into the state flow graph and successfully passed this task.

None of our human subjects used site search. This is most likely because the tasks they were given were very obvious: there was never a true need to resort to search. It also conforms with previous research that humans rarely use site search unless absolutely necessary. However, we

did test UseJax's search capabilities seperately. We discovered that UseJax requires very site specific information to be able to take advantage of its ability to input search terms. In particular, UseJax must be manually told what the search field is labeled. When search is set up properlty and given good search terms, UseJax almost always found the target state. In fact, UseJax could solve all four tasks listed above when told to use search instead of trying to simulate human behavior by following links.

Setting up search requires a bit of manual configuration, and most humans do not take advantage of search functionality. Thus, we consider search to be an ancilliary function of UseJax: nice to have but annoying to set up and not completely relevant to its main goal of approximating human behavior.

5 Limitations

UseJax is effective at small websites, but it posesses three major limitations. We briefly discuss these limitations below.

5.1 Non-clickable Interactive Elements

AJAX is very powerful and allows a wide range of interaction. UseJax only captures click-based and search-based interaction, but does not handle other forms of interactivity. As an example, a website could display a message after a fixed amount of time. Over 93% of user interaction with websites is click-based or search-based [2], so we feel missing the remaining 7% is an acceptable loss.

5.2 Depth Limit

During the crawl phase, UseJax attempts to create a state flow graph of the entire website. It is incredibly time consuming to try to index an entire website. Instead, we implement a depth limit. If the shortest path from the start page to a page is greater than the depth limit, the page will not be crawled and will be excluded from the state flow graph. Thus, if the optimal path from the start state to the target state is long, the search may give the erroneous result "no path found" instead of the true path.

We do not feel this is a major concern. The Law of Clicks shows that users are very unwilling to follow a long series of links. Research shows most users will give up on completing a task after following three clicks [1]. We feel most well-designed pages should not require a long list of clicks to reach important information. If absolutely necessary, the depth limit can be increased, although this will substantially increase the time required for the crawl phase.

5.3 Branching Factor

A larger concern is dealing with websites with a large number of clickables on a single page. If a site has an average of k clickables per page, then the crawl phase requires storing k^d states, where d is the depth limit. Site4 encountered this problem: there were thousands of states that needed to be crawled. At this time, sites with large branching factors are temporally infesable.

One potential solution is to preform a more targeted crawl. UseJax attempts to store the entire state flow graph. Instead, a new usability tool could use heuristics to focus the crawl and only store a portion of the state flow graph. The drawback of this method is that you lose the benefit of the two-phase approach. A new crawl would have to be preformed to test a new heuristic, instead of testing all heuristics on a single crawl.

6 Conclusion

Usability testing is a wide field with a variety of nuances and qualitative measures; it seems unlikely we will ever be able to entirely automate the process. Whether it is enjoyable to browse around a site seems like something we won't be automate without particularly advanced AI. What we can do is automate some of the smaller, more quantitative measures. With UseJax we looked at whether content was findable by a limited number of clicks. There are other static evaluation tools to see whether color combinations work or whether the time to load the page will be frustrating. If enough of these tools worked well enough, a robust, automated usability test program could be made.

For future work evaluating findability of AJAX sites we should explore different bases to build this tool on. Ironically, the problem with using CrawlJax is that it is too robust as a web crawler. We would prefer to work with something which allows more fine grained control over how which eventables it will crawl. This would hopefully make the time to find target states more manageable for websites with large branching factors. To improve user approximation, some basic natural language processing would be helpful. For a website findability tool, the most useful language processing to include would be categorization (to know whether a keyword is a person, for example) and synonym detection. It would also be helpful to be able to consider the CSS of the webpage: when users can't find a link relevant to what they want they click around easily visible links until they can find a new relevant link. A web crawler like CrawlJax with a built in decision tree might make the perfect base for such a tool.

References

[1] Joshua Porter. Testing the three-click rule. *User Interface Engineeering*.

[2] Ed H Chi, Adam Rosien, Gesara Supattanasiri, Amanda Williams, Christiaan Royer, Celia Chow, Erica Robles, Brinda Dalal, Julie Chen, and Steve Cousins. The bloodhound project: automating discovery of web usability issues using the infoscentπ simulator. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 505–512. ACM, 2003.

[3] Jeff Sauro. Search vs. browse on websites. *Measuring Usability*, 2012.

[4] UKOLN. Usability issues for ajax. Technical report, 2009.

[5] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012.

[6] F Ferrucci, F Sarro, D Ronca, and S Abrahao. A crawljax based approach to exploit traditional accessibility evaluation tools for ajax applications. In *Information Technology and Innovation Trends in Organizations*, pages 255–262. Springer, 2011.

[7] Natalia Negara, Nikolaos Tsantalis, and Eleni Stroulia. Feature detection in ajax-enabled web applications. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 154–163. IEEE, 2013.

[8] Swarthmore computer science homepage. www.cs.swarthmore.edu. Accessed: Dev 2013.

[9] Swarthmore dash. https://secure.swarthmore.edu/dash/. Accessed: Dev 2013.

[10] Swarthmore college homepage. www.swarthmore.edu. Accessed: Dev 2013.

[11] U.S. Dept. of Health and Human Services. *The Research-Based Web Design & Usability Guidelines*. U.S. Government Printing Office, Washington, enlarged/expanded edition edition, 2006.

[12] Peter Moriville. User experience design, 2004.

[13] Jeffrey Rubin and Dana Chisnell. *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*. Wiley Publishing, Inc., Indiana, USA, 2008.

[14] Chris Pilgrim. An investigation of usability issues in ajax based web sites.

Who Tracks the Taint Tracker?
Circumventing Android Taint Tracking with Covert Channels

Dane Fichter and Jon Cronin

December 20, 2013

1 Abstract

Modern smartphones are responsible for a growing number of functions in our daily lives, and consequently hold an increasingly large amount of sensitive data. Many mobile operating systems alert the user to each application's use of sensitive data upon install, but cannot prevent applications from using this data in malicious ways. The research community is aware of this threat, and has developed information flow tracking software to detect misuses of private data. However, this software is not foolproof. A clever attacker can leak private data from an application in an undetected manner using techniques such as application collusion, data obstruction, or a combination of these methods. Our research evaluates the effectiveness of techniques that extract sensitive data from phones and their ability to evade detection from TaintDroid, an information flow tracking software.

2 Introduction

The issue motivating our project is related to smartphone security, in particular relating to potentially malicious applications abusing Android's permission mechanisms. Smartphones allow users to download applications created by third party developers who present a potential risk. Particularly in the Android Marketplace, where applications are not thoroughly vetted for malicious intent, it is increasingly likely that apps are misusing sensitive information. Upon downloading an application, the user is prompted as to whether they would like to grant the app certain permissions, for example use of the device's microphone or the user's contact list. This layer of security is deceivingly reassuring for many users, and multiple papers have been written detailing how these permissions can be (and are in practice) circumvented to access and relay sensitive information to external servers.

Due to these known security threats, the research community has recently begun to try to ameliorate these issues. TaintDroid is an application that tracks the use of sensitive data across a mobile operating system, with the intention of finding phone applications that use this data in inappropriate ways. TaintDroid is a highly sophisticated application that catches misuse of data successfully, but it can be fooled by known methods of data extraction such as application collusion and implicit flow.

Our project is thus motivated by the existing research (see below) related to covert channels in the Android OS and detection of malicious applications, as well as real life security concerns we hope to highlight related to Android applications. In contrast to previous research, which identified general techniques for evading information flow tracking, we highlight specific ways that TaintDroid fails to recognize improper sending of this data and test the success of common evasion techniques. Furthermore, we propose defense mechanisms that could thwart some of these attacks.

3 Background

3.1 Covert Channels

Much work has been done related to malicious extraction of sensitive data from a smartphone. Some of these techniques are very subtle and work by reading information into seemingly innocuous data. Two papers show how to use accelerometer data to infer keyboard strokes and pin entries simply from the motion of the phone [2, 3]. These applications have permission to access accelerometer data, however the user is not expecting the data to be used in such a compromising way.

Similarly, Gasior and Yang show that one way to send this sensitive data to an outside party without being detected is to mask the transfer of this data with a steady stream of unrelated, non sensitive data to a server [7]. Specifically, they send a live video stream from the smartphone to the server, a process the user had approved, but they send each frame with a long or short delay. These delays are used to encode a list of contacts in binary form, with two short delays representing a zero and two long delays representing a one. The smartphone does not detect any unapproved data being sent anywhere, but the contact list is sent nevertheless.

Marforio, Francillon, and Capkun present a different way to send sensitive data: an "Application Collusion Attack" [9]. In essence, they launch two seemingly unrelated applications, one of which has permission to access sensitive data, one of which has permission to access the network. They use a covert channel to then transmit the sensitive data from the app that has that permission to the app that can access the network, which then sends it to the external server. Like the idea given in the above paper, the Android OS does not detect this breach of security. Furthermore, the authors argue that they can evade detection from information flow tracking software, such as TaintDroid.

3.2 Information Flow Tracking

Tracking an application's use of sensitive data to detect misuse of said data is a common tactic employed by researchers looking to report malicious applications. This is generally accomplished by putting a taint on a piece of relevant data which then propagates across the application. If an application tries to send out a piece of tainted data, the tracking software registers this.

This kind of information flow tracking has been done on desktops through the use of a whole-system emulation. TaintBochs works at the hardware level, tagging individual bytes that are deemed sensitive [4]. Panorama uses Google Desktop as a case study to accomplish a similar task. The authors write that their system yields higher accuracy due to the fact that their taint tracking is whole-system and finely grained [11].

TaintDroid applies these information flow techniques to a mobile platform, although without providing a whole-system emulation of Android due to performance concerns [6]. Instead, Enck et al make use of Android's virtualized architecture to "integrate four granularities of taint propagation: variable-level, method-level, message-level, and file-level." The novelty of their approach stems from the integration of these four techniques, as well as treating real-life performance concerns. The creators of TaintDroid argue that it can detect many instances of personal information being misused by even relatively popular applications. However, they concede that TaintDroid has limitations in terms of how much wrongdoing it can detect. In particular, they write that TaintDroid is ineffective at detecting implicit flow and the use of covert channels. We are investigating these claims with our project.

Another application that builds off TaintDroid is AppFence. AppFence uses TaintDroid as a base to track information flow within the Android OS, but also supplies "shadow data" to applications it suspects are attempting to extract sensitive information.[8] This shadow data can be a dummy variable or an empty file, and its purpose is to prevent the user's real data from being compromised.

4 Methodology

In order to test the limitations of taint-tracking software, we implemented numerous attacks which attempt to steal phone numbers from an Android phone. Specifically, we tested the ability of application collusion and covert data transmission to successfully transmit phone numbers to an external server without being detected by TaintDroid's taint-tracking framework.

Our tests were run on an Android emulator downloaded according to the instructions at Android's source code webpage [1]. The Android source code was compiled with the TaintDroid application built in, downloaded from the TaintDroid project's official website [5].

5 Evaluation

Each of these attacks was tested using an emulator running TaintDroid. To test each attack, we initialized the emulator, ran an application to automatically populate the contact list with randomly generated names and phone numbers, and loaded each attack into the emulator. For the attacks which did not employ collusion, we simply ran the application and determined whether or not its activity was detected by observing whether TaintDroid presented a notification. For the attacks employing collusion, we ran the principle application and ran the helper application after the principle application terminated. Again, we determined whether TaintDroid was able to catch the theft of sensitive information by observing whether or not TaintDroid presented a notification.

5.1 Control Attack

We began our experimentation by implementing a program which uses neither collusion nor covert data transmission to export contact information. The application uses permissions that allow it access to the wireless network and the user's contact information. When the application is downloaded, it reads all phone numbers from the contact list and packages them as a single stream, separated by new-line characters. This string is passed into a JSON package, which is then sent to an external server. As expected, TaintDroid detected this leak of sensitive information.

5.2 Covert Data Transmission Attack

This attack is contained within a single Android application, but uses a form of covert data transmission to elude taint-tracking software. As in our control attack, the application uses permissions which allow it to access both the networks and the user's contact information. The application reads in each phone number and covertly transmits each digit of the phone number to itself by changing the phone's volume and then immediately reading back the volume setting it just set. After reading each digit of a phone number from the volume settings in this way, the app bundles these phone numbers into a single string and exports this string to an external server via a JSON package. Due to the relatively unconventional method of transmitting data presented by this attack, we had anticipated that it would be able to decieve TaintDroid. Contrary to our hypotheses, TaintDroid was

able to detect this attack.

5.3 Implicit Flow Attack

In the paper explaining the implementation of TaintDroid, the authors admitted that a technique called implicit flow has the potential to foil their taint-tracking system. To test this hypothesis, we implemented an attack that uses implicit flow in an attempt to obfuscate its theft of contact information. The application uses permissions that allow it to read contact information and access the wireless network. The application reads in each phone number and implicitly assigns a value to a seemingly unrelated string variable based on the value of the phone number. The application then sends these phone numbers to our server. Despite the expectation of the authors that this technique would decieve TaintDroid, Taintdroid successfully detected this attack.

5.4 External Memory Attack

The external memory attack was designed to test TaintDroid's ability to maintain persistent taint tracking throughout the process of reading from and writing to external memory. The attack, contained within a single application, uses permissions that allow it to read contact information and access the wireless network, as well as read and write external memory. This application reads in all phone numbers from the contact list and writes them to a location in external memory. The application then immediately reads this data back in from external memory and sends it to our server. Despite the relative simplicity of this attack, TaintDroid was unable to detect the leak of sensitive information.

5.5 Collusion Attack

This attack uses collusion between two applications which share the necessary permissions for extracting and exporting sensitive data. The first of these applications uses permissions which allow it to read contact information and to write to the phone's external memory. This application reads in all the user's phone numbers, combines them into a single string, and writes this string to the phone's external memory. The second of these applications uses permissions which allow it to read from external memory and access the wireless network. This application accesses the file written by the first application and reads in the string containing the phone numbers before exporting this information to an external server using a JSON package. TaintDroid was unable to detect this attack.

5.6 Collusion and Covert Transmission Attack

The final, and most complicated, attack that we implemented uses both covert data transmission and collusion between applications to attempt to decieve TaintDroid. The primary application uses permissions that allow it to read contact information, change volume settings, and write to the phone's external memory. This application reads in the all of the user's phone numbers and re-reads them into memory using the same volume modulating pattern used in the covert data transmission attack. The application then writes the phone numbers to the phone's external memory. The second application reads the phone numbers in from external memory and exports them to an external server using a JSON package. As hypothesized, this attack was able to avoid detection by TaintDroid.

5.7 False Positive Attack

After observing TaintDroid's ability to detect out implicit flow attack, we hypothesized that we could easily induce false positives in TaintDroid. This attack, contained within a single application, used permissions to read in contact information and access the network. This program reads in all phone numbers from the contact list and branches depending upon the value of each phone number. However, this branching does not affect the final outcome of the program, which is simply a randomly generated string of integers which is sent by the application to our server. Despite the fact that we do not transmit the user's data, covertly or otherwise, to an outside party, TaintDroid still flagged this application as leaking personal information.

5.8 Results Summary

Attack Strategy	Apps	Permissions	Beat TaintDroid?
Simple	1	Contacts, Network	No
Volume Modulation	1	Contacts, Network	No
Implicit Flow	1	Contacts, Network	No
External Memory	1	Contacts, Network, External Memory Read/Write	Yes
Collusion, Simple	2	Contacts, Network, External Memory Read/Write	Yes
Collusion, Volume	2	Contacts, Network, External Memory Read/Write	Yes

TaintDroid was unable to detect data extraction techniques which made use of application collusion, a result we had expected. However within the context of one application, TaintDroid successfully registered data extraction even when obfuscated by volume modulation and implicit flow. This implies that TaintDroid's tracking is more zealous than we had thought, although the existence of false positives shows that this propagation is not completely accurate.

6 Future Work

In the course of testing our covert channels, we exposed some major weaknesses both in the Android OS and in TaintDroid's taint tracking implementation. Future work could work more closely with the source code of both the Android OS and TaintDroid to determine other lower-level ways to extract data, and could perhaps highlight other serious security flaws. Furthermore, we could test whether static analysis tools can detect that our applications are malicious. Based on the research we have accomplished, we now propose certain defense mechanisms that could potentially thwart attacks that use similar techniques to the ones we have implemented.

6.1 External Memory Protection

For our collusion attack, we write contact information to a file in external memory, which is then read in by another application. There are many ways for two applications to collude with one another, but to prevent this kind of attack, we propose more security permissions for applications trying to access external memory. For example, we can imagine a system where applications can only access files in external memory that they create, or one where applications are forbidden from accessing files created by other applications. There are certainly legitimate reasons for applications to create and write to files in external memory, but two applications accessing the same file seems like a flaw ripe for exploitation from colluding applications.

For taint tracking in particular, one possible way to better track sensitive data when it is written to a file in external memory would be to taint the file we are writing to. Maintaining a taint on the external file would allow TaintDroid to detect application collusion which makes use of this file.

6.2 Verifying External Network Accesses

Similar to more malicious malware, all of our attacks access an external server that receives the sensitive data we are stealing. One way for the Android OS to prevent data leakage of this sort is to verify that the server is not malicious. This is less relevant for our project as the server we set up is innocuous, however it is possible that by checking this website beforehand the OS could avoid interacting with a malicious server.

6.3 Permission Improvements

The fundamental problem driving this work is one inherent in Android's permission system. We believe that a more secure implementation of application permissions would make the kind of data extraction we performed much more difficult and obvious. For example, instead of having a blanket permission which allows access to a user's contact data, a permission that explicitly allows contact information to be sent over a network would better inform the user as to how the application is using their data. Similarly, Android could flag certain combinations of permissions as being high risk, such as personal information and any network permissions.

7 Conclusion

We had two main motivations driving our research. We wanted to determine which techniques are capable of extracting sensitive information from an Android phone without detection from taint tracking software and what specific factors determine whether taint tracking can detect our transmission. We were also interested in highlighting flaws in the Android OS itself, specifically related to misuse of application permissions. We demonstrated that taint tracking (as implemented by TaintDroid) has success catching cases of data extraction within a single application. Even with obfuscation techniques such as implicit flow and masking data within the physical features of a phone (specifically volume level), TaintDroid accurately identifies data theft. However, it fails to register attacks that make use of collusion between two applications and ones that hide sensitive data in external memory. Furthermore, we showed that TaintDroid may be overzealous in its taint propagation, as it is possible to trigger false positives within the system. Given TaintDroid's failure to catch several of our relatively simple covert channel implementations, it is apparent that any successful approach to combating this style of attacks must combine improved taint-tracking with improvements to Android's operating system.

References

[1] Android. Android source code, December 2013.

[2] Adam J Aviv, Michael E Locasto, Shaya Potter, and Angelos D Keromytis. Ssares: Secure searchable automated remote email storage. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 129–139. IEEE, 2007.

[3] Liang Cai and Hao Chen. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX conference on Hot topics in security*, pages 9–9. USENIX Association, 2011.

[4] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, pages 321–336, 2004.

[5] Enck. Taintdroid, December 2013.

[6] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 255–270, 2010.

[7] Wade Gasior and Li Yang. Network covert channels on the android platform. In *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '11, pages 61:1–61:1, New York, NY, USA, 2011. ACM.

[8] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These arent the droids youre looking for. *Retrofitting Android to Protect Data from Imperious Applications. In: CCS*, 2011.

[9] Claudio Marforio, Aurélien Francillon, Srdjan Capkun, Srdjan Capkun, and Srdjan Capkun. *Application collusion attack on the permission-based security model and its implications for modern smartphone systems.* Department of Computer Science, ETH Zurich, 2011.

[10] Clemens Orthacker, Peter Teufl, Stefan Kraxberger, Günther Lackner, Michael Gissing, Alexander Marsalek, Johannes Leibetseder, and Oliver Prevenhueber. Android security permissions–can we trust them? In *Security and Privacy in Mobile Information and Communication Systems*, pages 40–51. Springer, 2012.

[11] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.

STuneLite: A lightweight auto-tuning framework

Alex Cannon
Swarthmore College

Brian Nadel
Swarthmore College

Aashish Srinivas
Swarthmore College

0.1 Abstract

Shortcomings of conventional compiler optimization, due in large part to computer architectures growing increasingly more complex, has inspired an interest in tuning an application's algorithm-level parameters to optimize its performance. While such tuning continues to be done manually, automatic tuning (auto-tuning) methods are being explored as a more efficient and effective way of configuring applications. Such systems are useful for ensuring that performance-critical code can run at near-optimal efficiency on different system environments. Despite the potential benefit of autotuning, however, most current autotuning systems are restricted to specific applications or are difficult to use. Here we present STuneLite, an easy-to-use, universal framework for automatically tuning a generic piece of software.

1 Introduction

1.1 Background

Tuning is the process of modifying program parameters to optimize some measure (often runtime) on a given computer architecture. Many applications can gain significant performance boosts through tuning. Applications that use large buffer sizes, for example, often have an optimal buffer size that is dependent on hardware. Slightly decreasing this value can allow the buffer to fit into a cache, drastically improving memory access speed. More generally, tunable applications are distributed with parameters that work reasonably well on many platforms, but are less than optimal for any particular one. When performance is critical, as is the case with most scientific and numerical computing to name just one example, it is essential to find parameters values that are better than the out-of-the-box ones.

One example of software that can benefit autotuning is the GNU Multiprecision Library. This library contains approximately eighty tunable parameters that affect performance in varying ways, depending on hardware. The default values may be chosen such that they work reasonably well for a wide range of hardwares, and perhaps optimally for some particular machine. However, it's impossible for a single set of parameters to work optimally or near-optimally for all machines that run the application. Thus, to find optimal parameter values, one must somehow incorporate information about the hardware of the target machine. Much of today's software is still tuned manually, if tuned at all [4]. However, due to the complexity of many search spaces, tuning manually is generally time intensive, and the programmer can't find solutions as quickly or as well as an autotuner can.

Despite the possible benefits, a universal auto-tuning framework has yet to be developed and widely used. Easy access to such a framework could drastically lower performance, memory and other computing costs by closing what has been referred to as the widening gap between true peak performance and what is typically achieved by conventional compiler optimization [2]. Here, we contribute to closing that gap by creating a tuning framework that reduces the unnecessary cost of repeatedly observing, tweaking, compiling, and running an application by hand.

1.2 Motivation

Given the potential performance gains of tuning a piece of software, there have been several more-or-less successful research projects that have explored the automation of the tuning process: tweaking an applications configuration, compiling, running, measuring resource use, developing a new configuration to test and repeating over and over again. However, despite this research interest and the obvious potential for automatic software tuning, tuning continues to be done largely by hand. Even worse, many libraries come pre-loaded with arbitrary constants that have nothing to do with the systems they are running

on, and are not easily exposed for tuning (for example, *zlib* sets its buffer size to the arbitrary number 16384). We feel that there are two fundamental problems with existing auto-tuning platforms that has resulted in their lack of widespread adoption:

- generalizability
- ease of use

Most current auto-tuning frameworks are custom-made for specific applications. For example, signal processing libraries and sorting libraries have both been auto-tuned. However, these frameworks have almost no generalizability to applications outside their domain, and they tend to be difficult to use and configure. A more general framework could allow users to quickly and painlessly find near-optimal parameters for new applications that they create. Our goals for STuneLite were thus to:

1. allow tuning with very little modification of the source code
2. allow customization of the tuning process
3. minimize the time taken to adequately tune the program

1.3 Some Terminology

In general, there are three pieces of terminology that are important to understand. First is the concept of *search strategy*. In our implementation, a search strategy is a function that takes information about previous tuning runs as input, and outputs a new set of points to test on. Examples of tuning strategies vary in complexity from the random seletion of points in the domain (randoSearch), to more complex probabilistic methods [5]. The second important concept is the concept of a *parameter*. In general, parameters will be constants defined in preprocessor directives that will be declared once and used throughout the program. Some examples of parameters that can be tuned are buffer lengths, number of threads that are spawned, and perhaps even which library (of a predifined set of libraries) to use for a given task. The last important concept is a *sensor*, which is in general some measurement made after or while running an application. Important examples are memory usage and runtime. However, other conceivable sensors that could be tunded are processor utilization, cache hit rate, power utilization, or perhaps something more abstract like the number of requests handled by a server in a given time interval. Altogether, the goal of an autotuner is to search a parameter space for a parametrization (set of parameters) that will produce optimal or near optimal sensor values.

2 STuneLite Overview

The STuneLite library is designed to be highly modular. The system is broken up into five major components:

1. Application controller
2. Configuration file
3. Ad-hoc IPC
4. Search strategy
5. I/O manager

Parts 2-4 all work somewhat autonomously, and are tied together by the application controller, which directs the flow of data through the rest of the system as depicted in Figure 1. Before we look at each of the components of the system and how they fit together in more detail, we will discuss how our design choices help us achieve the three goals for STuneLite outlined in the previous section.

Our first design goal is to allow tuning to occur with minimal modification of the source code. This is an important goal because it makes the process more usable. In general, as described in the part about the Ad-hoc IPC, all communication between the source code and the tuning framework is done through the setting of environment variables. So, the user must only set the value of each variable in his or her source code to be equal to the value of an environment variable. For example, if we want to tune a parameter called *BUFLEN*, we would only have to replace the following line of code:

```
#define BUFLEN 16384
```

With:

```
int BUFLEN = atoi(getenv("BUFLEN"));
```

And then define a knob with name *BUFLEN* in the configuration file. No other modificaton to the source code is needed on the part of the programmer.

The second goal of STuneLite is to make the tuning process highly customizable. Specifically, we want the user to be able to easily customize the types of parameters that can be tuned as well, the search strategy that is used, and the sensors to measure. STuneLite comes built in with a few simple search strategies, and provides a simple way of designing new ones. The configuration file makes it similarly easy to customize the parameters that are tuned, allowing the user to define the a the range of values that the tuner should try. Sensors are generally measured by a shell script that by

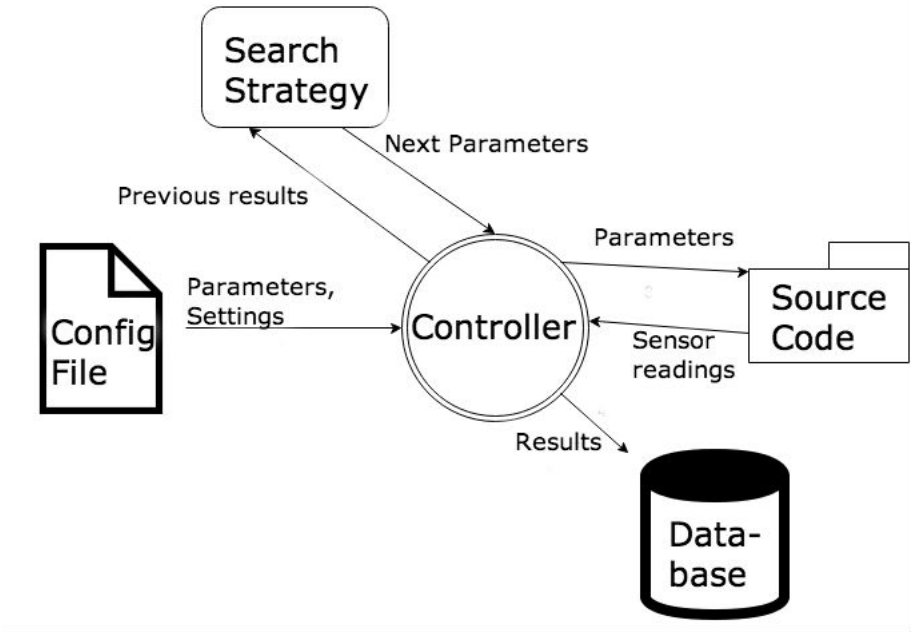

Figure 1: *Layout of STuneLite Framework*

default just measures the runtime and memory allocation of the application. However, it is easy to define other metrics (for example measuring the size of a compressed file or reading from the /proc filesystem) by creating another shell script.

The last goal of the tuner is to maximize the speed of the tuning process. Because we sometimes need to run the source application under a dynamic analysis tool like *Valgrind* (specifically if we want to gather memory allocation data), then each run can be much slower under STuneLite than outside of it. In general though, most of the speed of the tuning process has to do with how quickly we can get our parameters to converge on a local minimum. Although some of our methods are fairly naive (e.g. randoSearch, gridSearch), we have created a lightweight gradient descent algorithm . In our experience, this algorithm can be used to find local minima rather quickly on a variety of search spaces, leading to a speedy tuning process.

2.1 Configuration File

When using STuneLite, a user specifies information about the how to autotune an application using a configuration file. The configuration file is written in YAML, a format chosen because it is is both human-readable and easily parseable. The configuration file allows the user to define, among other settings, a pre-programmed tuning strategy to use, an objective function, the source application to tune, and (perhaps most importantly) the definition of the tuning knobs. The user must provide STuneLite with an environment variable that allows interface with the source application, as well as the range of values that the knob can take on, and its type.

2.2 Search Strategies

A search strategy defines the way an autotuner explores the search space defined by a set of parameters. STuneLite currently supports 4 search strategies: randoSearch, gridSearch, sequentialSearch, and hillClimber. All search strategies take the same input, which includes the number of iterations already completed, data about the knobs, and previous settings and results. A maximum number of iterations is also provided, and the main loop will cease calling the search strategy and return the best value so far if this number is reached. Some search strategies use all of this input information, and others use only a subset. All search strategies return a list of parameter settings for the main loop to execute before calling the search strategy again.

2.2.1 randoSearch

This simple search algorithm fetches a random value for each knob (within that knobs range) each time it is called.

2.2.2 gridSearch

This search strategy splits up the range of each variable into $p = \sqrt[k]{N}$ possible values, where k is the number of knobs, and N is the maximum number of iterations. We

then test every possible combination of those values, essentially creating a grid of the search space. Due to the exponential growth of the search space with the number of variables, this strategy samples the search space much more sparsely as the number of variables increases.

2.2.3 sequentialSearch

This search strategy tunes variables one at a time. Given a maximum number of iterations of N, sequentialSearch splits up the range of each knob into $\frac{N}{k}$ (k = number of knobs) values. It first sets all parameter values to their minimum possible configuration. Then it tests each of the $\frac{N}{k}$ possible values for the first parameter, while keeping the second parameter fixed. Next, our program chooses the value of the first parameter than yielded the best results, fixes that value, then repeats the process on the next variable. The strategy terminates when the final variable is tuned.

2.2.4 hillClimber

This search strategy is best described as pseudo gradient descent. The main loop finds the best set of parameters so far, and tests points around it. 2 points are tested for each variable, one where that variable is incremented and another where that variable is decremented (other variables remain fixed). As the program runs, step size decreases based on a linear function of the number of runs. Initially, parameters are incremented or decremented by 1/5th of their total range, and by the last run that value decreases to 1/20th of the total range. However, both these numbers can be easily changed, which may be useful if the user has some prior knowledge the search space. Increasing precision as the number of runs increases allows the program to find a region good solutions, then fine tune its solution afterward. The first point tested by hillClimber is determined randomly by calling randoSearch.

2.3 Ad-Hoc IPC

Rather than relying on more conventional methods of inter-process communication (pipes, signals, sockets, etc.), we have used a combination of files, database systems, and environment variables to be able to communicate between the source code and the tuner. First, environment variables (which must be consistent in both the configuration file and the source code) are used to reset the values of parameters between runs. Then, sensor measurements are relayed to the tuner through either the use of a file or a persistent database. The reason why we did not use a more conventional IPC to communicate between the tuner and the children it spawns off to run the source application is to increas the simplicity of the code, and to make the system easily portable between different platforms.

2.4 Input/Output Management

Each auto-tuning run (parameter settings and their corresponding sensor settings) is stored in two places: (1) in an SQLite database in case the autotuning history is needed for future tuning sessions and (2) on the heap in an array of arrays that is passed to the search strategy, allowing the search strategy to base its next move on past tuning results and also saving both the search strategy and the main tuning loop from having to extract these past results repeatedly from the SQLite database.

2.5 The Main Tuning Loop

The heart of the STuneLite framework is a lightweight loop that repeatedly tunes the source application with different parameters. The main loop handles the interactions between the other components of the system, and repeatedly tunes the source code with different sets of parameters. At every iteration, the loop gets a set of parameters to test for the user-defined tuning knobs by calling the search strategy and sets these parameters as environment variables that can be used by the source code. Finally, the loop forks off a new child process that runs the source applications with new parameters, and measures the values of various sensors after the execution.
The main loop contributes to the modularity of the system by allowing the user to easily define their own sensors. The default sensors that are measured are the execution time, and the amount of allocated memory (through the use of Valgrind). A new sensor can easily be measured by creating a shell script to feed into the program. For example, one could use another utility to measure the cache hit rate of the program, and use that as a sensor.

3 Evaluation

We ran STuneLite on two non-trivial test programs. The first program was a toy example defined by Rosenbrock's parabolic valley function. This function is interesting because it is easy to describe analytically, but has historically been difficult to optimize. We ran our autotuning methods on a sleeper program that waited for an amount of time determined by the value of Rosenbrock's function. Secondly we tuned a matrix multiplication progam, a well-known example of an tunable algorithm. can be assigned any value between 1 and the smallest dimension of the array, and performance often varies based on this value in interesting and meaningful ways.

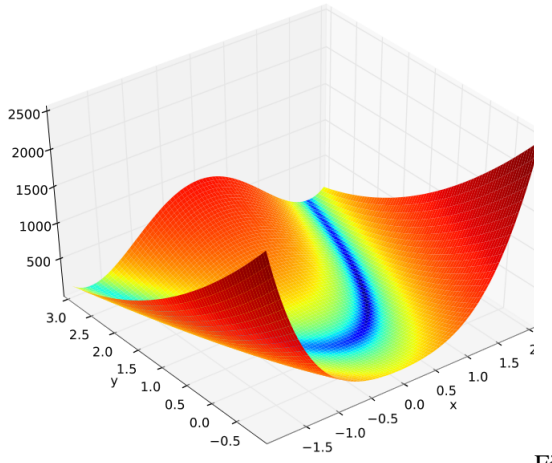

Figure 2: *Rosenbrock's Parabolic Valley*

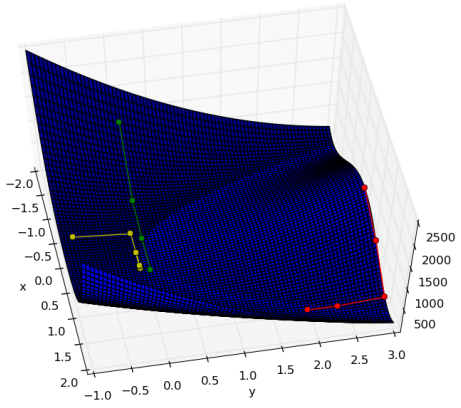

Figure 3: *Rosenbrock's Parabolic Valley with three runs of gradient plotted it*

With both test problems, we ran our autotuner on an AMD FX-620 Six Core Machine. The cache sizes and frequency of 'caper' are as follows:

L1 data caches: 6 *16 KB 4-way associative
L1 instruction caches: 3 * 64 KB two-way associative
L2 caches: 3 * 2 MB 16-way associative
L3 cache: 1 * 8 MB up to 64-way associative
Frequency: 3800-4100 Mhz

3.1 Rosenbrock's Parabolic Valley

Rosenbrock's Parabolic Valley is the function of two variables defined as:

$$f(x,y) = (1-x)^3 + (y-x^2)^2 \quad (1)$$

The region of interest for this function is $x \in [-2,2], y \in [-1,3]$, because this is the area immediately around the global minimum. The function is interesting because it is highly non-convex while also having a simple analytic form. The function obtains a maximum value of 2527 in our interval at the point (-2, -1). The function obtains a global minimum of 0 at the point (1, 1), but there are other points with very low value in the valley (1 at the point (0, 0) and 8 at the point (-1, 1)).

We ran our modified version of gradient descent on this parameter space (see figure 2). Clearly, the function converges to the parabolic valley within only 5 iterations. In fact, the largest of the 3 minima found was approximately equal to 4, meaning that all three of these runs found minima that were within .15% of the best value in the interval relatively quickly. Because each iteration requires 4 'checks' of points in the vicinity, convergence happened after about 20 total evaluations of the function. We consider this number to be reasonable, given the difficult search space and the simple search strategy.

3.2 Matrix Blocking

We also tested STuneLite on a blocked matrix multiplication algorithm. Traditional matrix multiplication can be very slow, as matrix values that will be used again in the future are frequently replaced from the cache. Matrix blocking is an attempt to increase temporal locality by essentially computing the partial result of a subset of the matrix (a matrix block), then combining those results at the end. Theoretically, a well-sized block is barely small enough to fit in one of the caches, allowing fast retrieval of elements of the block.

To visualize the search space, we ran an extensive gridSearch on the blocked multiplication of square arrays of dimensions 1000, 1500, and 3000 (figures 4-6). We ran the most tests on 1500 sized matrices, testing about 1 in every 4 possible block size values. About 1 out of every 50 possible block sizes was tested for the 1000 size matrix, while only every 250th value was tested for the 3000 matrix. Interestingly, the best block sizes were always those that had large remainders when divided into the total size of the array. With the 1500 array, for instance, points sligthly greater than 300, 375, and 500 all yielded good results, while points slightly less than those values were much worse. While the reason for this is not obvious, the results are meaninful. Changing the block size from 497 to 501 resulted in a 2.2x speedup.

We performed three runs of our hillClimber search strategy on the blocked multiplication of 1000x1000 matrices. We varied the parameters of the gradient descent algorithm from run to run, such that some runs had larger step sizes than others. The program ran for 20 iterations

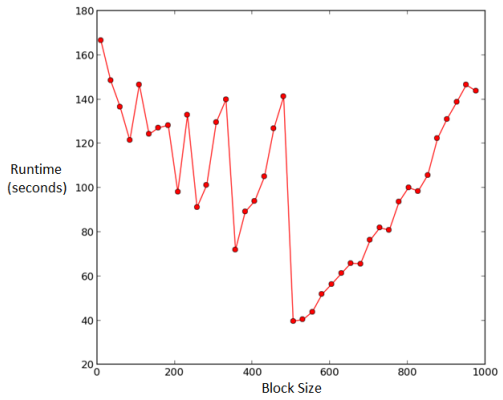

Figure 4: *Tuning 1000 x 1000 blocked matrix multiplication on Linux Ubuntu*

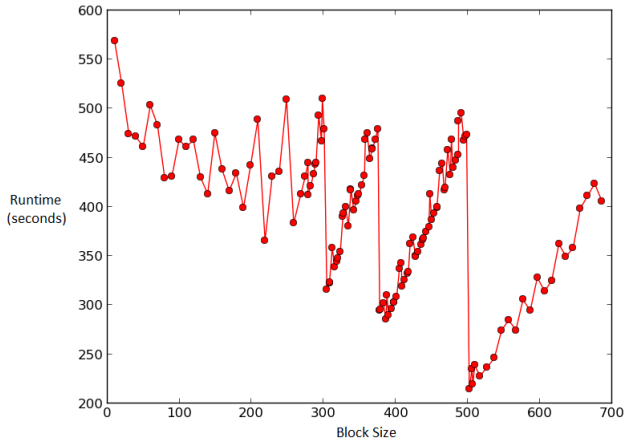

Figure 5: *Tuning 1500 x 1500 blocked matrix multiplication on Linux Ubuntu*

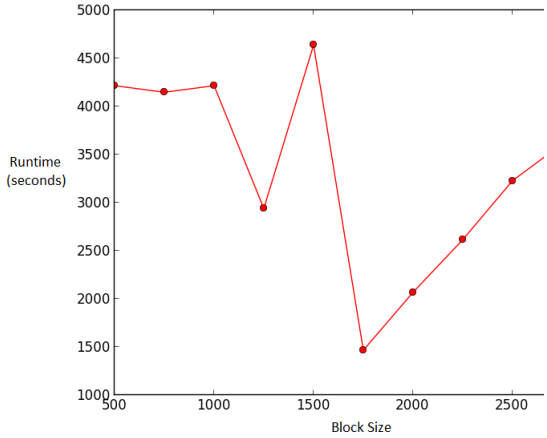

Figure 6: *Tuning 3000 x 3000 blocked matrix multiplication on Linux Ubuntu*

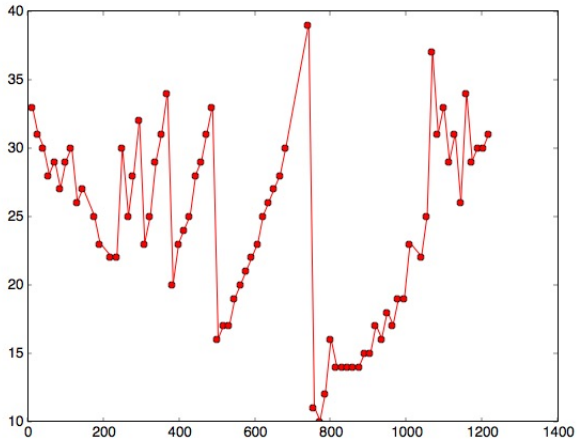

Figure 7: *Tuning 1500 x 1500 blocked matrix multiplication on Mac OS X*

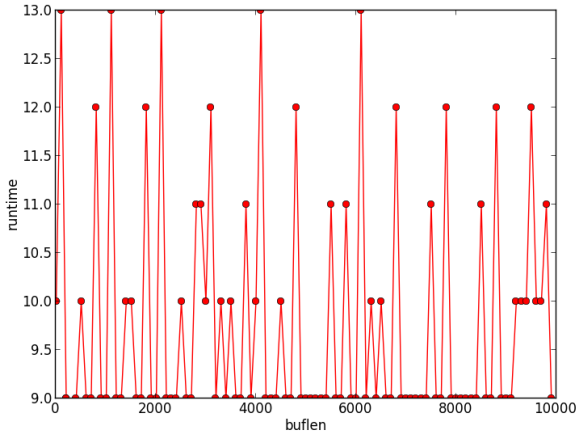

Figure 8: *Tuning the zlib compression library*

on all 3 runs. The first run had the largest step size, and returned a block size of 532. This value lies relatively close to the global optimum of approximately 500. The second run had a smaller step size, and returned a block size of 334, a local optimum and the second best global point. The last run had the smallest step size, and never reached on a local optimum. It returned a value of 634, though it was moving in the direction of the global optimum when the iteration limit was reached.

This is the kind of behavior we might expect from varying step size. Large step size finds the area of the global minimum, but hones in on the exact point poorly. A smaller step size is good at finding local minima, but not necessarily the global one. A very small step size may even move too slowly, and not reach any minima before iteration limit is reached. However, it's important to note that the observed results are due partially to chance (because of the random starting point), and typical behavior may differ than that observed in these tests.

3.3 Zlib Compression Library

To further demonstrate that STuneLite can be easily interfaced with a variety of applications, we attempted to tune the Zlib compression tool with it. Specifically, STuneLite was set up to tune the buffer length of the compression algorithm to optimize for runtime. The actual tuning results (see Figure 8) are more or less just noise, but indicate more the lack of sensitivity of the runtime of the compression to changes in buffer length (at least for the one file we were compressing) than any shortcomings of STuneLite. STuneLite, after all, can only tune the parameters its given against the performance metric its given to the extent that this metric can be tuned with these parameters. In some cases, as it appears in this one, there may not be any e STuneLite framework to Zlib was straightforward. Interfacing Zlib with STuneLite, on the other hand, was straightforward. Only minimal changes to Zlib's source code were required, and are nearly identical to those outline in section 2. The only other accomodation that needed to be made was a shell wrapper to be run in the main tuning loop that invoked the Zlib compression tool on a particular file.

3.4 Tuning Efficacy

The focus of our work was not the performance of the tuning itself, which has been explored to great length, but rather the ease-of-use and generalizability of the framework we developed. Nevertheless, we present results from several tuning sessions using relatively simple search strategies we implemented from scratch to prove the functionality of our framework, with the idea that the efficacy of the tuning itself will improve as more complex search strategies our developed for STuneLite.

To reiterate, the results here are shown simply to prove that the framework works, and are entirely dependent on the search strategies used, which as explained previously we designed to be entirely modular with the idea that users will develop more complex strategies afterwhich the tuning performance of STuneLite will rival those of other adhoc tuning programs.

3.5 Cross-Platform Compatibility

To demonstrate the cross-platform compatiblity of our system, we tested it within both Linux Ubuntu and Mac OS X machines. Specifically, Figures 2-6 are the results of trials performed on a machine running Ubuntu 4.6.3 on top of Linux 3.2, while the trial summarized in Figure 7 was performed on a machine running Mac OS X 10.8.5. The only modification needed to port STuneLite from Linux to Mac OS X was to change one line of code that measured run time, since the command line shells of each system handle time measurement slightly differently. This was a relatively easy fix, but in making it the resolution of the time measurement was reduced to a whole second, rather than microseconds, due to compatibility issues of the Unix date command on Mac OS X. This highlights difficulties in supporting multiple tuning "sensors" across different architectures, which we will adress again in the "Directions for Future Work section".

4 Related Work

Though a universal auto-tuning framework has yet to be developed, there has been other work towards this end. The closest effort to ours is the OpenTuner project, initiated by Ansel et Al. This project has a similar goal of creating an extensible, customizable auto-tuning framework [1]. The major difference between our strategy and theirs comes in ease-of-use. In general, their implementation offers added functionality at the cost of more effort on the part of the user to get their source code ready for auto-tuning. STuneLite attempts to be a much more lightweight framework that requires almost no modification of the source code and only a simple configuration file. We hope that this will make it easier to allow users to perform quick coarse-grained tuning on their applications. But beyond this, it is interesting to note that our system and theirs are incredibly similar in their high-level design, even though we were unaware of their implementation during the design phase of STuneLite. A particularly useful feature of their implemtation that would be very useful in STuneLite is what the authors

refer to as the 'configuration manipulator', which allows the set of parameters to be dynamic, so different parameters can be included and excluded from the system in the middle of the tuning process.

The other major work in this field, and one of the original motivations for this project, is ActiveHarmony, the framework designed by Hollingsworth et. al. at the University of Maryland [4]. This is a framework that, like STuneLite, is designed for "automated performance tuning" as Hollingsworth describes it, but with several key distinctions.

STuneLite and ActiveHarmony occupy distinct phyla within the Auto-tuning Kingdom. ActiveHarmony, in contrast with STuneLite, is an online tuning framework that continuously tunes an application as it runs. STuneLite, on the other hand and as we've described, is an offline tuning system that is meant to be run once to identify the optimal configuration for the application in its current environment that will be used for all subsequent runs of that application. Though several test runs of the application are executed within STuneLite's optimization loop, STuneLite is designed to ultimately "unhook" from its target application which will then run independently of any performance monitoring or configuration adjusting. ActiveHarmony on the other hand runs non-stop alongside its target application, never "unhooking" from it. The tradeoff here is flexibility vs. performance: while ActiveHarmony, by running constantly alongside its target application, is able to adapt the application to changes in its environment, it incurs a performance overhead in doing so. STuneLite, by contrast, assumes a more stable computing environment for its target application, avoiding any permanent performance overhead by ultimately unhooking from it after a sufficiently optimized configuration is found.

Lastly, it is important to compare our system with much more domain-specific systems such as SPIRAL [3]. Systems like SPIRAL are incredibly efficient at creating highly-performance optimized software in very specific realms (in SPIRAL's case DSP transforms). We believe that SPIRAL is in some sense a much more intelligent version of the sort of tuning system included with the GNU MP library, because it mostly relies on a massive amount of prior knowledge about its specific domain. We hope that STuneLite or systems like it will allow users to approach the tuning capabilities of a system like SPIRAL without having to actually develop such a system from scratch.

5 Directions for Future Work

Initial results for STuneLite are promising, though here we identify several additional steps to be taken towards the ultimate goal of an entirely generalizable framework. Since the focus of our work was to produce a proof-of-concept for a lightweight auto-tuning system, we did not focus on some of the usability issues. We hope that in future iterations, some of these issues will be solved.

The first area for improvement is the way in which STuneLite currently requires the user to modify their source code to replace parameters it wishes to tune with environment variables. Dynamic analyses could be performed on the source application to identify these parameters and replace them automatically. Though implementing such a solution was not within the scope of this initial project, it would prevent the user from having to modify the tuning applications source code, auto-tune, and then change the source code back, greatly streamlining the auto-tuning process.

Secondly, support for additional sensors needs to be added. Though runtime and memory allocation are surely two of the most import metrics users would want to tune for, there are certainly many others, like file size, cache hit rate, processor utilization, etc. While the user has the ability to create such sensors in the current implementation, it would be useful to have the more common ones built in. Moreover, methods for collecting a given sensor value may vary from system to system (as was the case with UNIX's date command across LINUX and Mac OS machines). Instead of collecting sensor values via command line shell utilities, for example, these values could be measured using other more universal third party libraries that we would then require the user to install prior to STuneLite to avoid these cross-system discrepancies.

There are limitless possibilities for the addition of new search strategies and the customization of supported ones. A simplex method could reduce the number of test points necessary to search many dimension spaces, while still producing gradient descent-like behavior. Probabalistic methods have produced promising results in the past [5], and are likely worth exploring. Modifying sequentialSearch may also yield interesting results. One could, for instance, perform many cycles of tuning parameters one by one, perhaps until a stable state is reached. Also, one could run sequentialSearch on more than one ordering of the parameters, and take the best result.

Our current implementation of hillClimber could also be expanded. Within a cycle (i.e. while testing $2\times$ numVariable points around a central one), our program is limited to testing points parallel to the dimensions of the search space. In other words, it can only change the value of one parameter at a time. When it reached the valley of good values in the Rosenbrock function, it was not able to parallel to it to find the global minimum farther along the valley. This may have been because the valley was not parallel to either of the two parameters. Future work may benefit from allowing hillClimber to test points that

change more than one parameter at once. This could be done to some extent while adding one more test point per cycle. One could use the first 2*numVariables test points to estimate partial derivatives, and then test one more point according to the relative values of those estimates. Adding more test points per cycle allows for more accurate prediction of the direction of greatest decrease, but has the downside of requiring more runs of the application. Other improvements to hillClimber might include random restarts and varying the step size or how quickly step size decreases.

6 Conclusion

STuneLite was designed to be a proof-of-concept to show that a small, lightweight system could be created to provide a general auto-tuning framework. The system we created was able to, with limited modification of the source code and the inclusion of a minimal configuration file, tune any piece of software. More importantly, we wanted it to be easy to add new sensors and search strategies to our system. Therefore, we created standardized interfaces to allow users to customize their own ways of measuring application performance and traversing the search space.

Our eventual goal for STuneLite is to make it a library that can ship with applications that have tunable parameters. Perhaps end users who have downloaded some software can have the option of either having a 'fast' install that just uses pre-defined parameters, or a 'tuned' install that runs STuneLite on the application for a small number of iterations. This would help to make the practice of auto-tuning programs a lot more widespread, and help decrease the gap between peak and realized performance.

7 Acknowledgements

We would like to Thank Ben Ylvisaker for all of his help and guidance over the course of this project. Not only do we owe him for his helpful advice, but for the name of the entire STuneLite framework as well. Our work presented here is merely our own version of a project first envisioned by Ben.

References

[1] Ansel, J., Kamil, S., Veeramachaneni, K., O'Reilly, O.M., and Amarasinghe, S. November 2013. OpenTuner: An Extensible Framework for Program Autotuning. MIT CSAIL Technical Report.

[2] Cohen, A., Donadio, S., Garzaran, M.J., Herrmann, C., Kiselyov, O., and Padua, D. September 2006. In search of a program generator to implement generic transformations for high-performance computing. Science of Computer Programming, 62(1):2546.

[3] Pschel, M., Moura, J. M. F., and Voronenko Y. 2005. SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation. 93(2):232 275.

[4] Tapus, C., Chung, I.-H., and Hollingsworth, J. K. November 2002. Active Harmony: towards automated performance tuning. In Proceedings of the IEEE/ACM Conference on Supercomputing, Baltimore, MD.

[5] Ylvisaker, B., and Hauck S., June 2011. Probabilistic auto-tuning for architectures with complex constraints. Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era.

AppFence: The Next Generation

Leah Foster
Madison Garcia
Samantha Goldstein
December 19, 2013

Mobile devices follow users everywhere, and so do third-party advertisers. Whether users know it or not, by downloading mobile applications they become advertising targets. Popular applications take sensitive information about users from their devices, including their device ID and location, and send that information to advertising and analytics servers. More often than not users are not informed of this, or if they are, then it is hidden away in vague and jargon-heavy Terms and Agreements. Our work gives users the ability to control the flow of data from their devices. Current systems have already made great strides, including TaintDroid, which labels instances of data misuse, and AppFence, which blocks access to sensitive, "tainted" data. We build off the AppFence model and improve some of its basic functionality to send out false information whenever possible.

1 INTRODUCTION

Consider an advertising agency seeking to identify a user's interests for targeted advertisements from data collected by the apps on the user's phone. Such an organization, upon obtaining the data from advertising and analytics (A&A) servers, stores user information and various pieces of metadata. Though this use is legitimate, it is very easy to identify a unique digital fingerprint from location data or social network data [3, 6]. Of course, the smartphone user can simply put the phone in a drawer and never turn it on, and it will consequently send no data.

However, we live in a world where this is rarely a viable option. And yet by using phones and other mobile devices, and by downloading apps, users risk their

privacy and security to exploitative applications and malware. An application might communicate with an A&A server in order to collect private data from a device and then build a profile of where the user goes, identified by their phone number and device ID (IMEI), and so on. But by faking sensitive data at the source, the servers will have obtained the wrong coordinates, phone number, and IMEI; the real information is out of reach. By consistently providing false information to A&A servers, we lessen the privacy risks of smartphone use.

We looked into TaintDroid and AppFence, two models that examine privacy, which we will detail more in Section 2. TaintDroid labels all sensitive data with a 'taint' to detect when the data is being sent away, whether maliciously or not. AppFence builds upon this platform by adding two techniques in particular– exfiltration blocking (preventing tainted data from being sent away to A&A servers) and faked data. AppFence will also blacklist servers that are known to be A&A servers.

Our policy modifies AppFence with a "guilty until proven innocent" model, so that apps receive false information by default. This is an improvement on AppFence, because applications may employ workarounds to evade taint detection. It is also beneficial because our system does not incur the slowdown created by TaintDroid and AppFence. Furthermore, if an application does need permissions for legitimate reasons (e.g. a maps app requesting location), then the application can be "whitelisted." This whitelisting model allows users to maintain functionality of the mobile device and still take control of personal privacy, by making such fine-grained decisions for themselves.

2 BACKGROUND

As Android grows in popularity, it has seen a blossoming of malware development and adaptation to new protections [13]. Malware frequently disguises itself as legitimate software so that users will install it, or employs other dubious methods. It puts a lot of effort into avoiding detection; in the best case, detectors only notice about 80% of malware [13]. Once on the phone, malware may subscribe to premium-rate services by sending SMS messages in the background and then blocking incoming SMS messages requesting confirmation [13]. Many aggressively seek users' data. But many non-malware apps, even top-of-the-market apps, also seek user data through sometimes questionably-legal channels, and that is what our research focuses on [4].

Grayware is substantially different from malware. First, these apps are less outright malicious–they do not financially drain their victim or disable the device. Second, they usually preserve some plausible deniability: if they collect user data and send it to remote A&A servers, they are simply following through on the end-user license agreement that the average user never reads. In many cases, information harvesting is how apps stay free. It is a common assumption and possibly implicit agreement that users are willing to give away some data so long

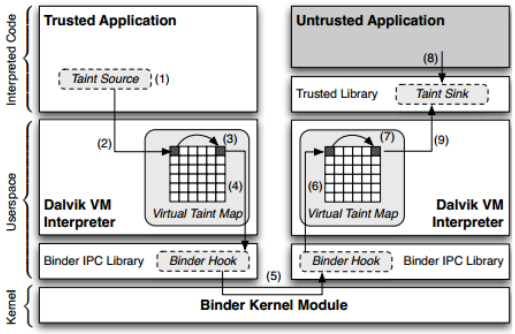

Figure 2.1: TaintDroid architecture within Android [4].

as they have access to the latest developed games. However, not every user wants to participate, and this is what we will focus on in our improvements to AppFence.

2.1 TaintDroid

Dynamic information flow tracking is a general term for tracking whether and how sensitive information is leaked during program execution [12]. The TaintDroid system is one example of information flow tracking specific to Android. TaintDroid detects when sensitive mobile data (e.g. phone number or location) is sent off the phone during application execution by applying a "taint" to sensitive data and tracking the taint [4]. At a high level, TaintDroid modifies Dalvik, the Android VM, to tag data as tainted when it is retrieved from a sensitive source. For example, accessing camera data returns camera data with a bit of extra metadata indicating the presence of a taint that the app does not know about.

TaintDroid tracks this tag from the source to a suspicious sink–usually when it is sent over the network–as seen in Figure 2.1. In its regular operation, TaintDroid then notifies the user that their information has been leaked, and logs the leak as well. All of this dynamic analysis occurs during runtime, and incurs a 14% slowdown. This slowdown factor is of course a hindrance, but the user only experiences slowdown on the order of milliseconds [4].

Several types of information are particularly desirable, especially phone information, the phone's unique identifying IMEI, and location data [4]. But TaintDroid merely identifies and logs breaches of privacy without attempting to combat them, and indeed is not adequate for protecting a phone against attacks [10].

2.2 MockDroid

The application MockDroid works to protect the user's information, using no taint-tracking at all [1]. MockDroid also modifies the Dalvik VM, and does so to wall off the phone's resources. Instead of real access to some phone resource

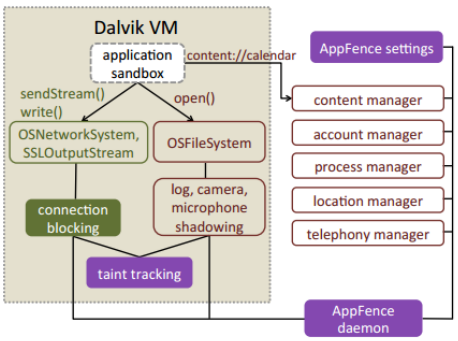

Figure 2.2: AppFence system architecture [7].

like the camera, an app will be told it is empty or unavailable. A smartphone user can allow an app to access the real resource, such as location data, internet connectivity, IMEI, and phone information like contacts. Many apps, though they may robustly handle loss of a resource, perform at reduced functionality when the information they receive is mocked [1]. As MockDroid was developed for Android 2.2.1, it is likely that more modern apps running on modern Android would suffer even more from loss of internet connectivity, for instance.

2.3 AppFence

AppFence concentrates more on app functionality. Implemented on top of TaintDroid, AppFence watches tainted data and prevents apps from accessing real resources if the use might be suspicious. AppFence works in two ways: exfiltration blocking and shadow data. When exfiltration blocking is enabled, if an app is trying to send away sensitive data over the network, AppFence will prevent the app from sending data, either by telling the app that the data was sent but actually dropping the packets, or by sending back a realistic error indicating that the network is inaccessible, similar to what the app would receive if the device were in airplane mode. When shadow data is enabled, AppFence will return entirely fake data to the app. For simple data, such as location or phone number, AppFence returns a hard-coded fake value. For example, when an app requests the device's location when shadow data is enabled, AppFence returns the coordinates of Google's Mountain View headquarters [7]. For more complex data types, such as contacts or calendar, AppFence returns an empty database cursor. Figure 2.2 shows how exfiltration blocking and shadow data are implemented inside Dalvik.

Testing AppFence depended on judgments of reduced functionality: in testing several apps, the authors examined the GUIs to determine if ads were present and if any normal functionality had changed. They also performed a few tasks with the apps to test for reduced functionality. The results were fairly good; most

apps performed up to par, but disabling network access still induced loss of functionality. [7]

Along with the slowdown, AppFence faces some problems because it is almost entirely inaccessible. Foremost, it is written for the now extremely outdated Android 2.1; second, installation requires rooting the phone. The latter problem exacerbates the former: since AppFence doesn't lie on top of the OS but directly modifies it, it becomes difficult or impossible to port and make forward-compatible. Even if AppFence were available for in-use Android versions, the rooting requirements are prohibitive for those who want to preserve their warranties. Using taint-tracing means that rooting is necessary, but updating AppFence and making it actually available is possible.

3 Our Idea

We present our project as evidence in our proposal that AppFence can and should be improved, first and foremost to work with current Android devices. The substance of our intended AppFence improvements would change its structure. The modifications fall into three areas: minimizing reliance on blocking network traffic (exfiltration blocking), improving fake data flow, and implementing a whitelisting policy to allow only approved apps to access real data.

3.1 Minimizing exfiltration blocking

With taint-tracking, AppFence currently identifies a suspicious use of tainted data and takes action. It may simulate lack of network access in order to block data transmission (exfiltration blocking) or replace real data with fake. The end modification would make this policy simpler and stricter. Taint-tracking has several weaknesses: an app may appear to legitimately acquire data, and then send it to an A&A server later on, or an app may work around taint tracking by assigning variables through control flow. This is alongside the danger from malware that can easily beat it [10], and the slowdown it incurs.

The decreased use of exfiltration blocking will allow some apps to have increased functionality when compared to the previous version of AppFence, as increased network access by apps, even with false data, will prevent apps from hanging or crashing due to poor handling of network failure.

3.2 Improved fake data

Eliminating exfiltration blocking puts the burden on fake data to prevent apps from acquiring sensitive data. While AppFence makes use of fake data, it uses a very simple implementation of hard-coded data; the authors admit that this is the bare minimum of plausible data. There are two main issues with this implementation. First, oversimplified fake data may not offer a realistic and invisible user experience. For example, if a user doesn't want to share their contacts list,

AppFence returns an empty database. Without having at least the illusion of contacts, the user may not be able to experience the full functionality of the app (or some preview of it). Second, such simple shadow data makes it easier for apps to attempt to determine if a user is running AppFence (or something similar) and then react by denying access to the user or otherwise undermining AppFence, similar to how many websites respond to adblockers [8].

We seek to improve the quality and plausibility of shadow data to counteract both of these issues. More intelligent shadow data will offer the user a better approximation of apps' full functionality, both for testing out an app before giving it real data and for using an app anonymously. For example, improved shadow location data can make use of the user's actual location to provide a random, false location within a reasonable radius of the true location. This would give full or near-full functionality to some apps that only really need an approximation of user data but request a fine-grained data. Additionally, if a user is comfortable sharing their approximate location with an app but not other information, this kind of improved shadow data offers the user protection when using apps that require more permissions for installation.

3.3 Whitelisting policy

Ideally, AppFence would work on a whitelist paradigm. By default, apps would receive full network access, but would only receive shadow data. In a guilty until proven innocent model, the user could decide whether an app should be allowed access to real data, like location or phone information. This forms a secondary permissions system that is modifiable at runtime and will not disable a running app. With these modifications, taint-tracking becomes far less necessary. This paranoid system reasons that all information sinks are at least a little bit suspicious, so why distinguish any of them? Better to not need to track, speed up operation, and protect all data at the source by only releasing it with explicit permission.

Naturally, these modifications can't fully address the issue of privacy versus functionality. Inevitably, some apps will fail to work as expected in small or large ways due to a lack of data. At this point, it remains the user's responsibility to determine whether remaining anonymous or attaining full functionality of an app is more important when the two inevitably come into conflict.

4 Evaluation

At the end of our project, due to several technical roadblocks, we determined that it would be best to simulate the results of updating and improving AppFence as we described.

First we can address the speed of operation. Though TaintDroid's reports do provide useful information, we decided to remove the taint-tracking from AppFence, leaving only the fake data. As we mentioned taint-tracking incurs a slowdown of 14% [4]. Though it may not seem like a hindrance, the slowdown negatively impact

performance in intense computing situations–especially on a low-powered device like a smartphone. Removing taint-tracking would simply remove that slowdown: if the system isn't doing the work, it will not slow down.

Removing exfiltration blocking also, as mentioned above, would improve apps' regular function. AppFence caused several apps to break, when it used exfiltration blocking [7]. Without the exfiltration blocking, AppFence would run more invisibly. If a tool like AppFence is less obtrusive to the user, then the user is less likely to unwisely turn it off. An important element of privacy is the user behavior, and AppFence should promote good, private user behaviors.

The whitelisting model makes user behavior a priority. The defaults, in such a model, are set to "very private," and the user is informed that changing the defaults will make their device and information less private. Thus they must make an active choice to reduce the privacy protections on their device.

The fake data sent away would also be improved, although the necessity of this particular improvement would have to be illustrated with further research. Location, for instance, is a pseudo-randomly generated set of GPS coordinates, not a static location or offset from the user's real location. Since it is unclear what the more malicious apps and A&A servers might do with the user's data, we aim to generate data for the worst case.

Our work modifying AppFence and Android source indicates that porting AppFence features to Android 4.1 or later would mean many subtle changes, and would still require rooting the phone and flashing AppFence onto it. Further work in updating AppFence would take this into account and try to make it easier for the user to acquire.

While that further work is still very much necessary, in our project we made good exploratory progress into AppFence, CyanogenMod, and the Android source. In continuing this project, we would expand the exploratory work into extensive development and implementation.

5 Conclusion

The work of AppFence and our improvements are an important to the lives and privacy of all mobile phone users. Though the thought of advertisers having information about our whereabouts is not immediately distressing for all, it is clear that the potential ramifications of such available data on every phone-toting American is problematic. Though advertisers themselves are not necessarily direct threats, our data is being poured into servers where it can be revisited at will. This means that our data is not only accessible by data-churning computers, but by other people as well, and this is where the potential deception can occur. Consider the recent revelation of the governments procedural wire-tapping and spying. On a corporate level, advertisers know an enormous amount about us, where we go, where we work, who we call, etc. This can be equated to what the NSA captured about the lives of "potential terrorists"; however, some members were

not as interested in national security, but rather used their government clearance to track the location and internet history of love interests [5]. When using mobile devices we are equally susceptible to the malicious consequences of the people behind the servers. And while the threat does not seem immediately pressing, it is very real. AppFence and the improvements we have made would allow us to take privacy into our own hands.

REFERENCES

[1] Alastair R Beresford, Andrew Rice, and Nicholas Skehin. MockDroid: trading privacy for application functionality on smartphones. In *HotMobile 2011: 12th Workshop on Mobile Computing Systems and Applications*, 2011.

[2] Cheryl Conner. Your privacy has gone to the [angry] birds, May 2012.

[3] Yves-Alexandre de Montjoye, Cesar A Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific Reports*, 2013.

[4] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 255–270, 2010.

[5] Siobhan Gorban. NSA officers spy on love interests, August 2013.

[6] Kieran Healy. Using metadata to find Paul Revere, June 2013.

[7] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[8] Dave Lee. Is it ethical to block adverts online?, December 2013.

[9] Kevin Poulson. Edward Snowden's e-mail provider defied FBI demands to turn over crypto keys, documents show, October 2013.

[10] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *SECRYPT 2013: 10th International Conference on Security and Cryptography*. SciTePress, 2013.

[11] Mu Zhang and Heng Yin. Transforming and taming privacy-breaching Android applications.

[12] Xiangyu Zhang. Dynamic program slicing and information flow tracking.

[13] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

Cross-Platform Testing and Analysis of Web Applications Using Selenium

Kenny Ning, Sam Zhang, Antonio Farias

December 20, 2013

Abstract

This paper presents automated testing frameworks for various popular applications on Firefox and Chrome. Music streaming app, Spotify, and Google's email application, Gmail will be examined. For each application, we write an automated series of actions using Selenium that navigate through the app's different features. Statistics such as speed and network calls are collected during these test runs. Though Chrome does seem to perform slightly faster than Firefox, it is difficult to make a confident statement about the performance differences between browsers.

1 Introduction

In recent years, web applications have become an increasingly popular platform for new products. Services such as music, data management, and image editing have popular implementations on the web that have both made it convenient and easy to perform fairly complex tasks. Thus, it is necessary to examine closely how we can effectively test web applications and analyze our results.

From a developer's perspective, testing and analyzing the different components of a web application can be time-consuming and tedious. Though there do exist built-in browser tools that help developers analyze the functionality of their application, they still require the developer to manually perform actions and navigate through the app to get results. To avoid these cumbersome tasks, we present an automated testing framework that automatically performs a predefined series of actions on a particular web application and collects statistics after the test run. This process is both essential to effective software development and can potentially provide some novel insight into the underlying workings of a given application.

In addition, the market of web browsers has increasingly become more diversified. Google Chrome, Mozilla Firefox, Safari, and Internet Explorer represent the four most popular and well-known browsers. However, even out of these four, it is still unclear as to which browser captures most users. The performance differences between these browsers has consistenly been a heated topic of discussion. Visualizations such as in [3] seem to merit lots of attention, as users

are constantly in search of finding the most efficient browser for their needs. A cross-browser testing framework such as ours could prove useful in future discussions of browser comparisons.

2 Background

Many frameworks exist that test browser functionality, automate tasks, or test a particular web app. We looked at different frameworks like Selenium, built by its creators for the purpose of running tests on specific applications [4]. Another framework, Koala, was designed specifically to automate business processes that needed to be done in web apps every day [2]. Firebug/Netexport allows users to export all of the logging that Firefox development tools normally do to be analyzed later [1].

Yet for all the diversity of frameworks we found, one noticeable omission was a tool for specifically comparing the performance of a web application across different browsers. As we read more about the increasingly fragmented browser landscape, in which no browser can claim a majority of users, we realized that such a tool would be of great use. We then set out to build a framework to test this idea, by piecing together many of the existing tools.

3 Concept

The original inspiration for our project came from exploring with different developer tools available in the Chrome and Firefox browsers and identifying differences in how an application behaved depending on the particular browser. Using these tools, we were able to glean fairly interesting insights about how modern web applications work. For example, we could see a timeline of events that the application went through to load the page, view network calls, and assess the different Javascript calls. However, playing around with these tools required a fair amount of manual work (i.e. clicking through buttons, recording the whole process), and a natural extension would be to automate the tasks of navigating through the web application. After that automated test is written, it would also be desirable to have the test export some statistics (e.g. the timeline, network calls) so that we could examine how the application performed after the fact.

Ultimately, we are hoping to identify interesting implementation details about these applications that were not immediately obvious upon first usage. Finally, we would like to compare the performance differences of these tests on the different browser platforms.

4 Methods

We decided to analyze two different applications using this process: Spotify, the popular music-streaming application, and Gmail, Google's email management

system. Spotify is a Sweden-based company that provides a service for people to stream music on-demand. It currently has more than 24 million active users, and we will be examining the increasingly popular web version of Spotify. Gmail is the most popular web-based email service, claiming over 425 million users. Because of the massive popularity of these applications, we felt that some testing and analysis of these apps would be informative and beneficial to an especially large number of users. Additionally, we use these applications on a daily basis and had a personal interest to learn more about their intricacies.

Custom tests were written for each application using the Selenium Webdriver for python (integrated with the unittest module framework). These tests were then run on both Firefox and Chrome. Statistics for Firefox were collected using the Firebug extension. Statistics for Chrome were collected by tweaking options in the Chrome webdriver.

In this section, we will discuss the details of how these tests were written. All experiments start out by logging into the application with a uniform password and login. Additionally, most of our testing frameworks incorporate the usage of explicit waits. This ensures that most asynchronous data elements have fully loaded and also serves as a way to naively mimic the non-instantaneous movements of a real-life user.

4.1 Gmail

Gmail has over 425 million active users worldwide, and has become the standard for personal webmail. Testing it and understanding how it works is therefore to the benefit of many. Two Gmail tests were performed after login: One calculating how long it took to open a compose message window (*open_msg*), and another on how long it takes to compose a message and then send it (*compose_msg*). These two tests are particularly interesting because they test Gmail's AJAX capabilities, media rendering (displaying the compose window and alerts) and network calls to send a message using HTTP.

4.1.1 open_msg

open_msg waits for the inbox to load, retrieves messages, loads links, and waits for the Gchat window to populate. Once all these elements are loaded, the test opens a compose window, which pops up in the lower right corner as a new iframe and is called via Javascript. Several of these functions are handled using AJAX calls, and continue to run even after the test is complete. Some examples include timers that fire every once in a while and seem to be scanning for something.

4.1.2 compose_msg

compose_msg acts once the compose window is loaded. It navigates through the compose window, initially having to focus on the "send to" field. After typing in the recipient email, it must focus on the subject field, skipping out

of focus for other recipients, before focusing on the message field and then the send button. After activating send, the application waits for a fixed amount of time for the message to send. It is important to note that messages may not send in a fixed amount of time, but examining the network calls later on might give us a sense of this. This test is implemented using ActionChains, which queue up key presses, and has Selenium perform them in one continuous call upon calling perform(). This is analagous to storing all our actions in a buffer and executing the entire buffer. The user can also observe that the message is sent asynchronously in a thread, and that the user can keep navigating the app while the message is being sent.

4.2 Spotify

Spotify is one of the premier music streaming services that offers a wide library of songs and helps users organize and share their music on the cloud. After carefully inspecting the source code, we can see that Spotify is generally organized by different iframes, where each iframe is generally responsible for a different feature (e.g. Spotify radio, Spotify search). In addition, most data is handled in an asynchronous fashion, where images, messages, and other data populate the data non-deterministically. While this allows for smoother and more natural user interactivity, this model may prove to be a hurdle as we implement our tests, as it is very difficult to detect when a page or a particular element has "fully loaded".

Within the SpotifyTest class, we decided to write three smaller unittests that tested three different iframes: *test_discover*, *test_follow*, and *test_radio*.

4.2.1 test_discover

test_discover simply waits for the home page to load, which currently happens to be the Discover application. This is a relatively new feature that suggests different artists or songs that the user should listen to based on previous listening history. This test simply waits for a given period of time for the data to load. It also provides the application some time to load the different navigation icons that will allow us to move to the different iframes.

4.2.2 test_follow

test_follow clicks on the Follow navigation button and loads the corresponding Follow feature. This feature is the primary social component of the Spotify application and allows users to track their friends, artists, or organizations/labels on the service. Therefore, we felt that at least ensuring the social driver of the service loaded properly was an important component to our test. While this feature specifically tests the follow feature, one can very easily tweak the source to test any other feature in the service.

4.2.3 test_radio

test_radio loads the Radio feature and plays a song from the default radio station. The Radio feature allows users to craft a "radio station" based on a particular seed (artist, track, album, or genre) that then continues to play music similar to the seed.

This test was the most difficult to implement because of the way that Spotify encodes its different iframes. Each iframe id was generated uniquely at runtime, so it was impossible to switch to an iframe based solely on id. This made it difficult to actually find the play button we needed to click. Thus, we ended up searching for this radio play button by iterating through all available iframes. Despite this brute force approach, we were successfully able to play a song from Spotify Radio for a few seconds.

4.3 Data Collection Tools

We used the Firebug and NetExport extensions to collect data in Firefox. The NetExport extension is an extension of Firebug that allows network data to be exported automatically. This allowed us to run tests written in Selenium while measuring first response, overall network times, and latency between items downloaded. In our evaluation, we discuss overall network times, which was derived by summing the blocking, waiting, connecting, and receiving sub-times given by NetExport.

In Chrome, we spoofed the header so that the browser presented itself identically as in the Firefox condition, and we toggled the DesiredCapabilities flag in the Chrome WebDriver to turn on performance logging. This flag was unfortunately unavailable in the Firefox version. The performance logs contained more diverse data than the NetExport logs. It included rendering times, Javascript execution times, and also network times. For the purposes of comparison, we filtered through this data to extract the network call times. These network call times were derived by subtracting the time that specific browser objects raised a Network.loadingFinished event with when they raised their Network.requestWillBeSent event. This bracket should contain all the subtimes given by NetExport, such as time spent blocking, waiting, connecting, and receiving data. Intermediate events (Network.dataReceived) were dropped because they lacked a comparison in Firefox.

5 Evaluation

We will be evaluating our testing frameworks using two fairly coarse-grained metrics: Selenium unittest runtimes and network times. Table 1 and Table 2 break down the Selenium runtimes for Gmail and Spotify, respectively, which is the amount of time the test took to finish. As with all our tables, average times will be reported in bold face. The network times metric will be further analyzed using two finer-grained interpretations of what network times actually mean.

The first network time evaluation will directly compare overall network times across browsers with the results from the Selenium unittest times. Overall network times are derived by summing up the time it takes to request and receive each individual element in the browser. This effectively "unrolls" the asynchronicity of the browser list, and we hoped to examine how effectively browsers compress data into asynchronous calls. The Selenium unittest runtime is provided as a way to normalize these network times, since their respective scripts sometimes required subtly different timing mechanisms.

The second metric is the comparison of specific network call times across browsers. By looking at how long it took for Chrome vs. Firefox to download individual items, we could determine how much variance there was with downloading each individual item, and determine the relationship between download time variance and file size. This provides us with information on 1) whether Chrome or Firefox is more consistent with providing steady download times and whether either is inherently faster, and 2) whether the unpredictability in download speeds occurs more with heavy, high-filesize media objects or small asynchronous calls. In other words, are the fluctuations in download speeds mainly caused by browser or network latency, and how are they handled differently by browser?

Because the network analysis did not give incredibly promising results, we decided to only discuss the network analysis in the context of the Gmail testing framework.

5.1 Selenium Unittest Runtimes

Table 1 and Table 2 display the unittest runtimes for the Gmail and Spotify unittests, respectively. Three trials were taken for each individual test and then averaged. At first glance, it seems that Chrome passes through the unittests more efficiently than Firefox does.

While these times do suggest a small speedup in the unittest runtimes for Chrome, it is slightly more difficult to make the logical leap from automated testing runtimes to real-time user runtimes. The actual actions of a user are fairly unpredictable, while the actions in our testing framework are completely deterministic. Though we do include some basic sleep functions to mimic the non-instantaneous actions of a user, one should still interpret the numbers and evaluation with caution.

5.2 Network Experiment 1: How tightly are asynchronous calls packed across Chrome and Firefox?

This experiment was set up to test whether asynchronous calls were performed more efficiently in Chrome vs. Firefox for Gmail. Looking at raw Selenium data is unreliable because the two browsers required different amounts of built-in timing/explicit waits. The asterisk in the title of Table 3 reflects the fact that the presented Selenium data has been offset by the different timing constants.

The Chrome test time was subtracted by 5 seconds and the Firefox version was subtracted by 20 seconds, reflecting the sum of timing events built into the tests.

If we look at the ratio between network times/unittest times provided in Table 3, we can get a rough idea of how efficiently a particular browser packs network calls into a given time unit. From this metric, it seems that Chrome is more efficient in its network calls, since it has a higher ratio. However, our results are likely a quirk of Firebug and the extra overhead it produces. A low-level performance logging flag as that built into Chrome invariably incurs less overhead than dynamically loaded extensions such as Firebug and NetExport. Because of these timing differences, it is hard to analyze the data directly. In order to determine the exact nature of this variation in measurement, we performed a second experiment.

5.3 Network Experiment 2: How reliable and valid is browser network call data?

This experiment was performed to understand the possible confounds in experiment 1. In order to test the performance differences across browser, we had to make sure that the performance metrics were equal. This was done by comparing elements that were downloaded by both Firefox and Chrome during their experiments, as logged by Firebug/NetExport in the case of Firefox, and the WebDriver performance logs in the case of Chrome. The trials varied in the number of elements that were downloaded, and the number of elements that were downloaded in common between the two browsers. However, Table 4 shows that Chrome tended to have more downloads than Firefox during the test period. Since the Firefox test ran nearly 60% longer, it is unlikely that this is due to background asynchronous calls occuring after the browser has idled.

Only certain elements were common across trials. This is likely due to changes in session data embedded into the URL of GET requests. In general, elements with the word "static" somewhere in the URL were consistent. Even though the second URL contained a lot of "gibberish" data, it was still consistent across trials.

Our comparative data also did not provide conclusive results. As Table 5 shows, the mean download time for what was presumably Javascript associated with Google Talk was much higher in Chrome than in Firefox. This seems to be in direct contrast with our conclusion in the first network analysis experiment, where Chrome appeared to be more efficient. Unfortunately, this may suggest that the data collected by summing up Firebug data was not directly comparable to data gleaned from the Chrome performance logs.

6 Challenges/Future Work

Perhaps the most obvious continuation of our project would be to extend our testing and analysis frameworks to more web applications. Spotify and Gmail

Table 1: Gmail Unittest Runtime Results

Firefox		Chrome	
open_msg	compose_msg	open_msg	compose_msg
11.902	23.199	9.237	18.330
11.720	20.759	10.451	19.513
10.532	20.820	10.125	19.264
11.385	**21.593**	**9.938**	**19.036**

Table 2: Spotify Unittest Runtime Results

Firefox			Chrome		
test_discover	test_follow	test_radio	test_discover	test_follow	test_radio
12.194	6.944	15.341	11.040	5.330	13.682
12.400	7.018	14.339	11.000	5.513	13.155
12.285	5.820	14.639	10.987	5.264	12.507
12.293	**6.594**	**14.773**	**11.009**	**5.369**	**13.115**

Table 3: Selenium Runtimes vs. Total Network Call Times on Gmail*

Firefox			Chrome		
Selenium	Network Calls	Ratio	Selenium	Network Calls	Ratio
14.224	20.896	1.469	16.112	29.482	1.83
11.354	15.950	1.405	18.609	30.698	1.65
13.751	29.101	2.116	17.698	38.245	2.161
13.11	**21.982**	**1.663**	**17.473**	**33.110**	**1.88**

Table 4: Number of Network Calls on Gmail Tests

Chrome	Firefox	Both
154	124	54
155	112	46
171	137	58

Table 5: Number of milliseconds logged for downloading Google Talk JavaScript'

Chrome	Firefox
563	144
59	42
127	20
245	**69**

were selected as our case study applications particularly because of their popularity. However, it might also be interesting to examine other less popular applications, especially ones that make more intense CPU calls (e.g. image/video editing apps, games). These more CPU-intensive apps might log more significant changes across browser than what we had observed for Spotify and Gmail.

Unfortunately, much of these kinds of applications would be difficult to implement using Selenium. The Selenium Webdriver primarily drives action by finding particular HTML elements and interacting with them (e.g. sending a mouse click event, sending text). Thus, it was particularly difficult to meaningfully interact with an application whose actions were mostly driven by Javascript or Flash. Perhaps using a different testing framework more suited to these kinds of applications would yield us more in-depth tests. For example, CrawlJax was suggested as another possibile framework to use for AJAX-based applications.

Lastly, it might have been interesting to examine additional performance comparison metrics, such as the time it took for Javascript to render. Different browsers have different Javascript rendering engines, and considering how much scripting is used in the web applications we examined, there might have been some more noticeable performance differences with regard to this metric. Other interesting extensions could include how well the different browser caches improve performance, or timing how fast the different CSS elements take to render.

The challenges that we faced throughout this project also raise an interesting discussion point about the lack of standardization of web development testing tools. The main difficulty in comparing performance metrics across browsers was that Firefox and Chrome each had their own logging/debugging tools and would report data that initially seemed very different. Oftentimes, we would even encounter some interesting data in the Chrome logging that just did not exist in the Firefox logging. This suggests there is a gap in the currently available testing tools, and the presence of a more standardized data collection tool for web applications across browsers could greatly aid in the process of understanding cross-browser functionality. This concept is well suited enough to be its own research project that could have a positive impact on the web testing community.

7 Conclusion

We present a framework for testing and analyzing modern web applications. These tests could allow developers to ensure the functionality of their applications as well as understand more of what is happening under-the-hood. Overall, the applications performed pretty similarly across browser, though we did notice a brief speedup in the runtime of an application's unittest when it was running under Chrome. However, this speedup wasn't incredibly significant, and it was difficult to reconcile whether this speedup was truly due to specific aspects of the Chrome browser. On the network level, the lack of standardization of web development tools also made it difficult to come to conclusive results. Overall, understanding these kinds of testing and analyzing procedures is and will continue to become an essential piece to correct and make consistent software projects.

References

[1] Firebug Community. Firebug - web development evolved. `https://getfirebug.com/wiki/index.php`, 2013.

[2] Greg Little et. al. Koala: Capture, share, automate, personalize business processes on the web. `http://tlau.org/research/papers/koala-chi07.pdf`, 2007.

[3] Jacob Gube. Performance comparison of major web browsers. `http://sixrevisions.com/infographics/performance-comparison-of-major-web-browsers`, 2009.

[4] Antawan Holmes and Marc Kellogg. Automating functional tests using selenium. In *Agile Conference*, 2006.

Cstrace: visualizing strace

Cynthia Ma - cma1, Sola Park - spark1

December 19, 2013

1 Abstract

Trace data can be used for finding errors in processes, locating inefficiencies in software, and understanding the background interactions of executed code. However, the large and complex output returned from a trace can be difficult to analyze and understand. We propose Cstrace, a prototype of a trace visualization tool that visualizes output from strace, a system call tracing utility for Linux. Our technique focuses on reorganizing output by system calls, highlighting possible sources of errors, and suggesting relations between system call instances by using different display options that users can navigate through. Evaluation by user study supports our future goal of creating a visual learning tool for strace.

2 Introduction

Trace records information about program execution. Since a trace can include complex interaction with other software and the operating system, developers use different tools to understand and analyze the program's behavior. A software visualization is one of these tools. Software visualization is defined by Koschke [3] "as the mapping from software artifacts—including programs—to graphical representations". Some of the graphical representations include illustrations, diagrams, or graphs. Visualization for the purpose of conveying information gained from a trace has grown as a popular strategy for helping software engineers to grasp the behavior of large-scale programs and to gain insight into the possible problems of complex software. By using trace visualization, it is easier to communicate information about program behavior, like concurrent timing of methods in different threads, the distribution of processor time during exectution, or the memory use.

Among different kinds of utilities related to system trace, strace is a lightweight tool that traces system calls and signals. Run on the Linux platform, it allows the users to view a process and its interaction with the operating system. Although it is both commonly used and easily accessed, mainly for debugging purposes, its output requires that the users have a basic understanding of its syntax and commands to use the tool. Even for experienced users of strace, navigating the output to locate the source of error or the patterns between different interactions can be time-consuming. Put differently, this means that once the output can be organized in a way that can represent a pattern or path within the output, the users may find it easier to use strace regardless of their experience level with strace. We concluded that visualization would provide the means to do this. While there has been research in the visualization of trace outputs, there isn't yet a tool developed for navigating strace output. Therefore, we developed Cstrace, a prototype of an strace visualization tool. Our target audience are people familiar with some of the concepts in Computer Science but not necessarily with computer systems or trace tools.

Since Cstrace is a preliminary effort towards an application that can be distributed, we conduct a user study to analyze the efficiency and to recognize the potential of the program. The data collected from the study will guide our future work in improving our prototype to fit its purpose.

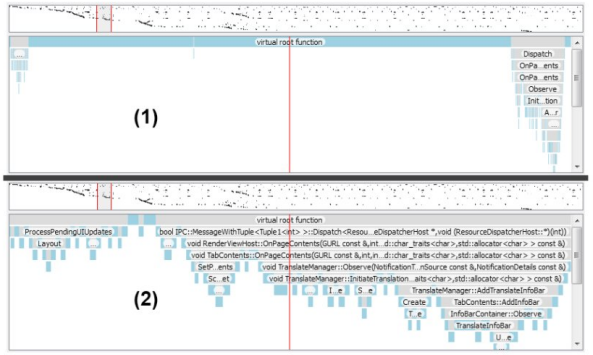

Figure 1: A screenshot of the thread overview and sequence view displays in the multithreaded software system visualization tool proposed by Trumper et al [4]. The two parts compare linear and logarithmic scaling of the sequence view.

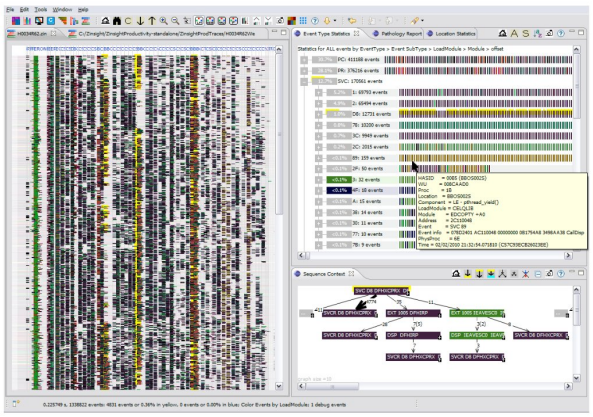

Figure 2: A screenshot of the Event Flow view (left), the Event Statistics view (upper right), and the Sequence Context view (lower right) of Zinsight [2].

3 Related Work

Traces can be used to serve various purposes such as performance analysis and optimization, debugging, and troubleshooting, with the sheer volume of data being its main limitation. In the process of mapping the original information gained from trace to graphical representation, it is possible that a visualization can become specialized for one of these purposes more than the others. For example, a multithreaded software system visualization tool proposed by Trumper et al [4] chooses this path (Figure 1). It is a tool with two windows of three organizational schemes that is aimed at enabling user comprehension of multithreaded software and its performance optimization. The first window, textual thread overview, allows textual searching through the trace for relevant threads. The other window contains visual thread overview(s) and sequence view(s). The visual thread overview is a general mapping of the events, and the sequence view is a more detailed display using call stack organization to highlight concurrency between threads. The sequence view can be adjusted on a linear-to-logarithmic scaling to allow real-time views of subsections within the program as well as an overview of the entire program organization. As such, this tool guides users to explore and navigate the interactions between multiple threads. However, by focusing on the performance optimization issue, it lacks the information needed for debugging.

Zinsight [2], on the other hand, provides users access to information needed for different uses of trace. Its three windows of different visualization schemes each display flow within the trace, event statistics, and sequence context, respectively (Figure 2). Its Sequence Context view in particular, contrasts with call stack representation, a method commonly used to represent calling hiearchies in previous existing Java program visualization tools and other tools like the above. Our tool was similarly developed to serve the multiple uses of trace such as program comprehension, debugging, and performance optimization, although the approach is more simplistic. Instead of three seperate windows, we chose to use a single view that can display different visualizations on user demand, with certain information conservable between display options such as our Error Highlight (Section 4.4).

VDP of lviz [5] is a tool proposed by Wu, Yap, and Halim for visualizing traces using an extended DotPlot visualization scheme. Applicable problems include software failure diagnosis, performance opti-

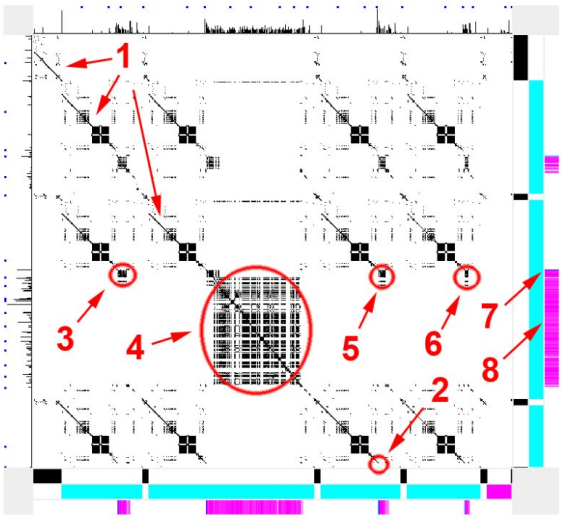

Figure 3: A screenshot of an extended DotPlot display created by VDP of lviz. Areas that indicated diversion between the two plotted traces are highlighted with numbers and arrows [5].

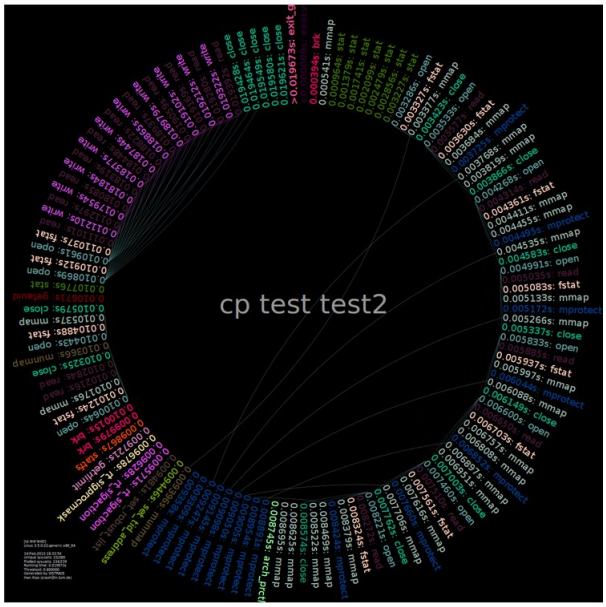

Figure 4: A graph output of vistrace on the command cp test test2 [6].

mization, and event comprehension. The basic idea is to set two traces along the x- and y-axis of a binary DotPlot to highlight regions of interest with resulting color mixes, large groupings of events, or a lack of symmetry (Figure 3). Unlike the previous examples of related work, this tool's graphical representation doesn't include any textual or numerical information from the trace. While this concept allows for more space efficiency in displaying the information on a single window, it creates a need for users to learn the possible implications of the components of the graph (histograms, colors, matching) back towards the actual properties of events from the original trace. Because our tool is designed to be accessible to new users of strace, we mostly chose to keep as much of the original textual output as we could in the different organization displays of our tool.

Previous support for visualizing strace includes vistrace [6] (Figure 4), which visualizes the output produced by strace -ttt cmd as a circular graph. We don't consider this a tool for navigating strace output, as with Cstrace. Vistrace focuses on presenting an image, rather than providing interactive features to analyze strace output.

4 Visualization

Based on our observations of how strace users generally organize or catagorize strace output information, we created several functionalities. With a display window on the right, these functionalities are listed as buttons or input boxes on the the left of our graphical user interface. In section 5, we will present an example to illustrate how these functionalities can be applied.

4.1 Command to strace

To start, our tool asks for a command to run strace on. (Figure 5-(A)) The normal strace output is stored and then parsed for use by the functionalities we define in the following subsections.

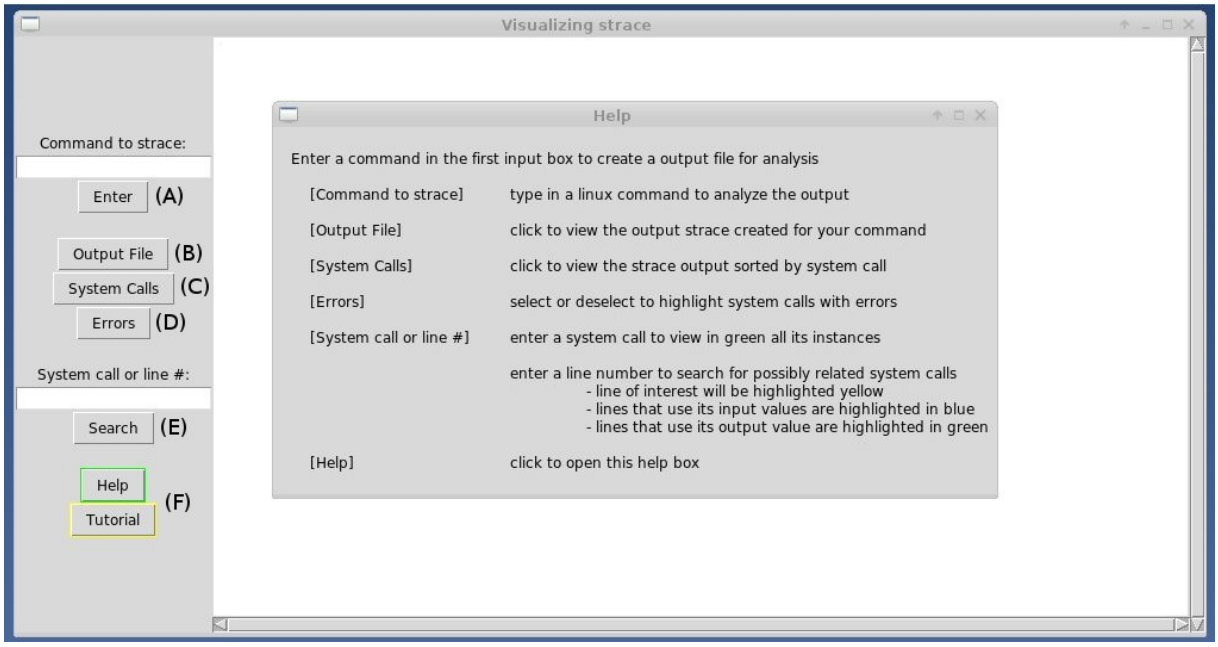

Figure 5: A screenshot of the opening screen for Cstrace.

4.2 Output File

Clicking the button labeled Output File (Figure 5-(B)) lists in the display window the same output that would result from the following terminal command:

strace -command-

On the left of each line is a boxed number that counts individual system calls. These numbers, unique to each system call instance, will remain throughout the use of different functionalities.

While this display doesn't change the organization of the original information, we included this functionality for contextual referencing.

4.3 System Calls

Clicking on the System Calls button (Figure 5-(C)) lists in the display window, the strace output sorted alphabetically by system call and then chronologically within each system call category. Each individual instance under a system call category will have its inputs seperated by commas, followed by a dividing tab space and its output. On the left of each instance are the same boxed numbers from the Output File display, though their ordering will likely be shuffled by the alphabetical organization of the system call instances.

With this display, we can see which system calls are called repeatedly, maybe significantly more than others. We can also view all instances of a particular system call more efficiently and locally.

4.4 Error Highlight

Selecting the Errors button (Figure 5-(D)) highlights warning instances in orange. While the Errors button is selected, highlighted warning(s) can be viewed in either the Output File display or the System Calls display. In this way, errors can be viewed in context or grouped under different system calls. This functionality was designed to warn users of error sources when debugging.

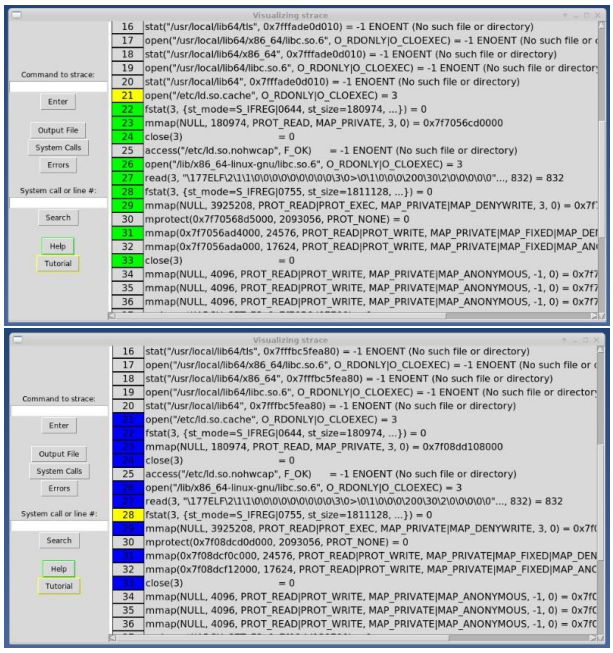

Figure 6: Screenshots of the Path Highlight functionality of Cstrace in use.

4.5 Search

Entering the name of a system call in the input box (Figure 5-(E)) labeled "System call or line #:" will result in the Output File display, with instances of the searched for name written in dark green instead of the default black text. In this manner, instances of a particular system call can not only be easily grouped but also viewed in context.

Search by line number is the Path Highlight functionality, mainly geared towards program comprehension (Figure 6). It highlights in yellow the boxed number of a searched for line and finds other system call instances that may be related to it. A system call could be related if it shares input parameters or output results. We chose to represent these two possible relationships separately; green highlights lines associated with the output result of the searched for line (Figure 6, top), and blue highlights lines associated with its input parameters (Figure 6, bottom).

In the first category, the tool searches for system call instances where the output result of the searched for line is being used as an input parameter. In Figure 6, the output of line 21 is a file handler "3," and the tool highlights the boxed number of any following instances that takes a 3 as an input. The use of this functionality is currently limited by false positives because of the possibility that the value of an output result is used for different purposes between system calls. For example, line 22 and line 27 both take a "3" as input, but the information referenced is different. The information referenced by the 3 input to line 27 is from the file opened in line 26. As such, all green-highlighted instances from line 26 on are actually not related to the searched for line 21. We therefore also highlight in green the lines with the same output result, such as line 26, where the information referenced by an output result may be changing.

In the second, the tool searches for system call instances where the input parameters of the searched for line is being used as an input parameter, or given as an output result. The figure shows an example, where the "3" input of line 26 is found as input or output in other lines. Again, there is the probability that the parameter references different data in between system call instances, so the user must manually define the relationships between system call instances hinted by this functionality.

Through this color scheme, we hope to convey a sense of path connections between different system calls that may be temporally seperated.

4.6 Help/Tutorial

Clicking the Help button (Figure 5-(F)) brings up a window with a short summary of the functionality available. Clicking the Tutorial button (Figure 5-(F)) brings up a window with a three-page tutorial that steps the user through this tool in solving a simple debugging problem, and introduces Path Highlight examples.

```
\#include<stdio.h>

int main(void)
{
   if(NULL == fopen("MyLinuxBook","rw"))
   {
      printf("\\n program failed\\n");
   }
   else
   {
      printf("\\n program successful\\n");
   }

   return 0;
}
```

Figure 7: Source code for the simple program, Open MyLinuxBook

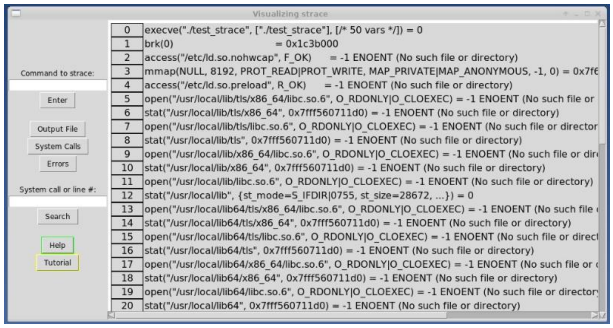

Figure 8: The display when "./test_strace" is entered into the Command to strace input box. Also the Output File Display

5 Example of Use

We present an example that is provided in our tool's tutorial, which illustrates how our tool can be used for debugging.

Figure 7 shows the source code of a simple program, "Open MyLinuxBook [1]." When this program is executed, it returns the simple message, "program failed."

This program is provided to users without the source code. Instead, they must use our tool to find that the error occurs because the file "MyLinuxBook" does not exist.

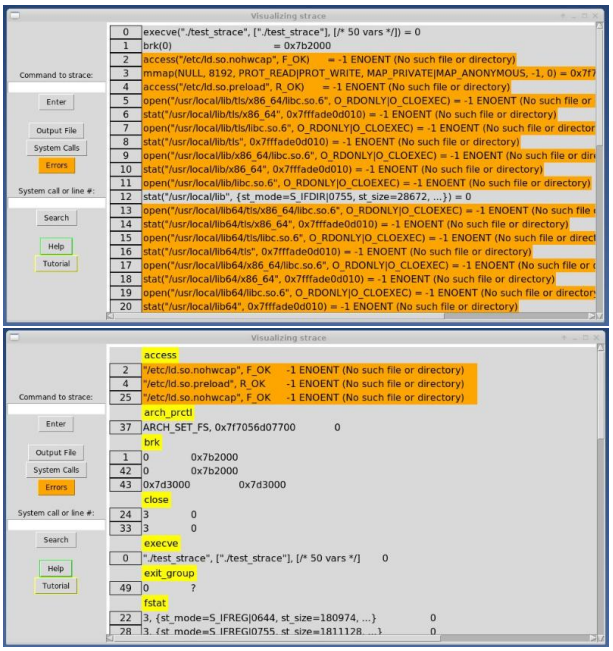

Figure 9: The Output File display (top) and the System Calls display (bottom) when the Error Highlight functionality is turned on.

Typing "./test_strace" in the input box of the program where it asks for "Command to strace" results in many lines of system calls in the display window to the right (Figure 8). Turning on the Error Highlight functionality by clicking the Errors button, we can navigate between the Output File display and the System Calls display to find possible sources of error (Figure 9).

Since the program's purpose is to open a file, we look at all warning instances of Open system calls by scrolling to the Open category listed in the System Calls display with the Error Highlight functionality turned on (Figure 10). There are nine instances of the Open system calls to examine. The last instance has the input "MyLinuxBook" and an error output that says "No such file or directory." At this point, we know that the reason why the program failed is that the file it is trying to open doesn't exist.

Another way to find warning instances of the Open system call is to search by system call name with the

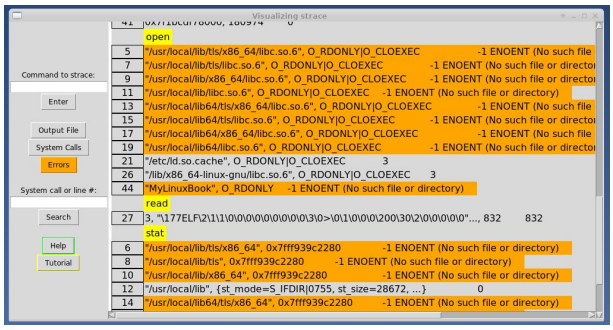

Figure 10: The open system call instances with Error Highlight on. Notice the last instance takes the file name "MyLinuxBook" as input.

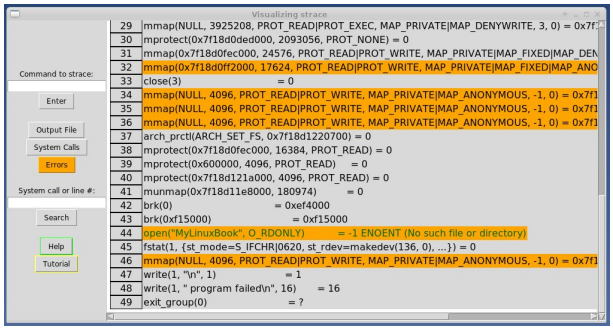

Figure 11: The error of interest written in dark green and highlighted orange.

Error Highlight functionality on. Typing "Open" in the input box labeled "System call or line #:" results in the Output File display with instances of the Open system call written in dark green. In this manner, the Open calls can be viewed in context (Figure 11). Again, at line 44, there's a clear explanation for the program failure.

6 Evaluation by User Study

Our primary goal was to build a visualization software for strace output, targeted towards people with little experience with strace. To evaluate our progress and guide our future work, we conducted a simple user study. Four undergraduate seniors, majoring in Computer Science, volunteered to use our tool for approximately 20 minutes and provided feedback. When asked, all four claimed to have had little experience using strace.

A difficulty most volunteers had was confusion over the Path Highlight functionality. This was expected since this is currently our least developed functionality and might not be intuitive for new users of strace. Suggestions for improving this functionality included the use of more graphical features, like arrows between related lines and the highlighting of actual inputs and outputs shared between system call instances.

Another suggestion was to "add a description of what each system call is doing" including a definition and the syntax for each system call (volunteer 2). This would be done in the System Call display to help new users unfamiliar with system call names, parameters, and behavior.

Other criticism was mainly limited to the tool's presentation, rather than the tool's functionalities. Examples include the scrollbar not responding to the mouse scroll wheel and the Enter button not responding to the keyboard's Enter key. While these issues are important to fix, we will not be listing them all here, as they are not specific to understanding the strace output our tool visualizes.

All volunteers agreed that our tool had the potential to be a useful tool for introducing strace to new users.

7 Future Work

From the user study evalution, the main improvement for our tool's immediate future would be to add a library of information on system call names, use, behavior, and output syntax. With this information, we can not only add the suggested descriptions to the System Call display, but also more accurately link related system call instances for Path Highlight. The Path Highlight functionality should be improved to be more intuitive without explicit instruction from the tutorial. We will explore different possibilities of visualizing Path Highlight, including a seperate display with extracted instances of system calls, which

are then linked with arrows with their shared information highlighted.

Other improvements for our tool would be geared towards overcoming our limitations with analyzing large traces, such as a trace of the python interpreter. When a large output file is generated for our tool, the visual display begins to break down as the graphic library used to generate our user interface cannot handle creating a text display for over 1500 lines. We plan on using a different graphics library to solve this problem. Another issue is the slow down of System Call display and Path Highlight as these functionalities currently require iterating through all system call instances and sorting them. We plan on solving this by changing the data structures used to store the lines of strace output or the parsing algorithm.

8 Conclusions

Cstrace, a prototype for an strace visualization tool, was developed for an audience who have limited background knowledge in system call interactions, but with a basic understanding of Computer Science or programming. Although strace itself does not require extensive understanding of system calls, it is difficult for new users to know where to start looking for sources of errors or program inefficiency without reading documentation on strace use. There is also little functionality in the way of helping users to better comprehend the program in its entirety. Thus, by using Cstrace, even users with limited knowledge of a system's implementation can understand a process's meaning and interactions. Cstrace is currently a prototype under development, but this and similar projects motivate further research into the use of visualization as a learning tool.

This paper summarizes Cstrace functionality, an example of use, and the user study conducted to evaluate its limitations and potential.

9 Acknowledgements

Special thanks to Benjamin Ylvisaker for his feedback and support throughout the research and development. Thanks also to Hugh Troeger, Bowen Wang, Yeayeun Park, and Hongin Yun for their participation in the user study. We are grateful to all our users for their feedback and their time.

References

[1] H. Arora. Linux strace command - a magnificent troubleshooter, June 2012.

[2] W. De Pauw and Steve Heisig. Zinsight: A visual and analytic environment for exploring large event traces. *SOFTVIS'10*, pages 143–152, 2010.

[3] R. Koschke. Software visualization in software maintenance, reverse engineering, and reengineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 87–109, 2003.

[4] J. Trumper, Johannes Bohnet, and Jurgen Dollner. Understanding complex multithreaded software systems by using trace visualization. *SOFTVIS'10*, pages 133–142, 2010.

[5] Y. Wu, Roland H.C. Yap, and Felix Halim. Visualizing windows system traces. *SOFTVIS'10*, pages 123–132, 2010.

[6] H. Xiao. Vistrace, February 2013.

Analyzing Android Malware and Current Detection Systems

Adrien Guerard and Danny Park

Abstract

Android security is a growing concern in the mobile industry as the number of devices in circulation continues to increase. Researchers and anti-virus companies continue to discover new malicious software (malware) on Android devices that strive to steal sensitive information and harm the device-owner in various ways. While most anti-virus softwares operate at the system level, recent research shows promising results for taking a network-level approach instead. Based on these promising results, we investigated a network-level detection system to gain a stronger understanding of the potential benefits and pitfalls. Additionally, we examined and reverse-engineered a recently discovered piece of malware called BadNews to see how the latest malware operates on the surface and at the implementation level.

Originally we had planned to perform clustering and signature analysis on an Android malware dataset we had received, but due to significant deviations from the model's assumptions and difficulties in trying to elicit malicious behavior, we decided to return to our original goal of improving and experimenting with our own version of BadNews: WorseNews. We created our own piece of malware for the purpose of investigating the capabilities and limitations of Android malware authors and anti-virus detection. Our malware is capable of penetrating the Android market in something called a "zero-day attack", in which one takes advantage of the fact that anti-virus systems utilize code signature methods for detecting suspicious apps, that are only effective at classifying known malware. Given the growing number of Android malware being released to the public each day, as well as our own experience developing and testing malware, it is safe to say that the Android platform is perfect for authors wishing to trick Android users into giving away their information and money.

1 Introduction

The Android operating system is growing in its share of the mobile market ever since its release in 2008. It currently owns 81% of the smartphone market while iOS only accounts for 12.9% [9]. Owning such a large portion of the market makes it a more lucrative target for malware developers. Android system APIs allow for a significant amount of access to sensitive data and user information,

as well as the ability to execute downloaded files at runtime. The Android OS uses a permissions system for handling which applications are allowed to use which resources. Upon installation, the user is asked to allow the application in question to be granted certain permissions. Studies show that many widely-used apps are over privileged, and that most individuals do not pay attention to what permissions they grant [1].

Many permissions are relatively harmless, but some allow for the application to act maliciously without being obvious or asking the user for any information directly. The simplest form of malware reads a user's sensitive information and transmits it over the network. Apps granted with permissions for the network and reading device information can transmit a user's IMEI number (specific to a device) as well as their phone number. Just these two pieces of information can be incentive enough for someone considering developing a malicious Android app when sold in bulk on the black market.

However, Google has taken steps to detect malware, and with the app needing to be pre-approved before going onto the Play Store, it is not easy to publish an obviously malicious app. There are certain permission patterns that look suspicious given certain types of applications, and applications that have sensitive permissions raise the most red flags. Recently a family of malware titled 'BadNews' was able to make it to the Play Store. It infected 32 applications and was downloaded up to 9 million times [8].

Detecting BadNews on a per app basis would be difficult given the fact that each app is different but contains the same ad network library. Desktop malware also tries to disguise itself as different programs and applications with some common malicious instructions. As Android malware becomes more prevalent, it will be necessary for their developers to start to obfuscate the malicious functionalities of their application. Android malware will most likely mirror desktop malware in the future, by constantly getting reused so as to continue publishing malicious content even after a previous app has been identified as being malicious.

Researchers have tried to get around this problem by developing techniques for extracting malicious network signatures from pools of malicious network traces organized into clusters by similarity of how they interact with the network. Any malware that steals users' information will most likely send the information over HTTP in a simple GET or POST. It allows the server to control whether or not the device should be actively malicious or not, which makes discovering all of an application's behaviors without looking at the source code much more difficult. However, by using network signatures, one could detect future or even other similar malware activity and stop the information leak at a network level.

Especially given the rising prevalence of smartphones, this technique could be used to detect malware once it has been identified. It also means that malware that simply reuses the same network communication will also be identified as malware, even if it is "unknown" as being malware. Although the device is still infected, one can at least hinder the collection of data on the user as well as render the device unable to receive instructions from the server.

We hope to further explore the argument that network-level analysis should

have a larger role in anti-virus detection and prevention. By cutting off the connection between host and client, much of the malware is rendered useless. Smartphones could therefore theoretically self-verify the safety of any HTTP request they make by inspecting it. Monitoring could also be performed by internet service providers, which would have the greatest impact in curbing the continued communication with command-and-control (C&C) servers, if done correctly.

2 BadNews

Lookout, a mobile security firm, discovered a new family of malware called BadNews in April 2013. The effects of the malware were widespread as it was downloaded up to 9 million times across 32 different applications on the Google Play Store [8]. We reached out to security researchers [6] and acquired an infected application (.apk) that serves as a live wallpaper to unsuspecting users. The list of permissions that the live wallpaper application requests should be a red flag to any individual. Ordinary live wallpaper applications do not require any permissions, but this application requests the following permissions:

- SYSTEM_ALERT_WINDOW
- RECEIVE_BOOT_COMPLETED
- INTERNET
- ACCESS_NETWORK_STATE
- READ_PHONE_STATE
- INSTALL_SHORTCUT
- VIBRATE
- ACCESS_WIFI_STATE
- WRITE_EXTERNAL_STORAGE

In order to analyze the infected app, we took a series of steps to decode the .apk into its Java source code. Upon examining the source code, we determined that the decoding process did not work flawlessly as there were blocks of code that did not make sense. For example, there were multiple instances of unreachable code below return statements.

Despite the decoding errors, we were able to parse the code and gain a detailed understanding of how BadNews operates. There are two classes that make up the majority of the BadNews codebase. The first class extends BroadcastReceiver and is triggered every time the device boots up or receives/sends a phone call. The class is tasked with determining if the main malware service is running. It starts the service if it is not running and does nothing otherwise.

All of the malicious activity takes place in the service. The service communicates with a command-and-control server located in Russia. The URL of the server, www.androways.com, is hardcoded as a static variable in the service. This particular server is located in Russia, but researchers have discovered that BadNews also has active servers in Germany and Ukraine [8]. When the service starts up, it registers an alarm (e.g. a function that emits a signal at regular time intervals) to go off every four hours. Every time the alarm goes off, the service sends a request containing device information to the server. The information includes device ID (IMEI, MEID, or ESN), phone number, phone model, and network operator.

The server responds to the device's request by sending a key which corresponds to a specific instruction the service should execute. For instance, when the service receives the 'stop' key, it cancels the alarm and discontinues itself. The service is responsible for executing the following ten instructions sent from the server:

- "news" - downloads a file from a location specified by the server and prompts the user to install it
- "showpage" - launches the browser and loads a URL
- "install" - downloads a file from a location specified by the server
- "showinstall" - seemingly similar to "news" but could be an error from decompilation
- "iconpage" - sets an icon on the device's homescreen that launches the browser and loads a URL
- "iconinstall" - sets an icon on the device's homescreen that prompts the user to install a downloaded file
- "newdomen" - changes the central server URL
- "seconddomen" - changes the secondary server URL
- "stop" - cancels the alarm and stops the service
- "testpost" - instructs the service to send another request to the server

BadNews behaves in a manner that makes it difficult to detect. It manages to avoid attention by performing activities for short periods of time in four hour intervals. This is to avoid detection systems that scan network activity intermittently. Additionally, the authors of BadNews obfuscate the service's activities by using response variables sent by the server as inputs to other functions. This makes it difficult to know what other servers the service is contacting through code examination.

3 Nework-level Detection

Perdisci, Lee, and Feamster [7] developed and tested a multi-step clustering of malicious internet traffic for the purposes of extracting a representative set of network signatures. Their first step is referred to as coarse-grained clustering, and is done by comparing simple statistics calculated about each malware sample's internet traffic: "the total number of HTTP requests, the number of GET vs. POST requests, the average length of URL's, etc." [7]. The second step is fine-grained clustering, which takes a closer look and applies string comparison algorithms such as normalized Levenshtein distance and the Jaccard distance of two sets of strings i.e. calculating the difference in parameter keys and values. Fine-grained clustering is only done between network traces of the same coarse-grained cluster. This is done to reduce the number of string comparisons performed, which are computationally expensive compared to the initial, coarse-grained clustering. They finally perform a cluster merging step in which they merge some of the fine clusters based on computed distances between cluster centroids.

A signature is formed by first creating a set of pools from which to perform the Token-Subsequences algorithm on. Initially selecting one of the malware samples to be the seed for that cluster (chosen at random), each pool is initialized with the one of the requests from the seed malware sample's network trace. For each pool, the closest (using the same string distance function as in fine-grained clustering) request from each unique malware sample in that cluster is added to that pool. Once all the pools are filled, a signature can be extracted from each pool using the Token-Subsequences algorithm which is described here [3]. It is a simple alteration of the Smith-Waterman algorithm used for finding the largest substring between two different strings. Once the signatures are formed, they are merged by measuring the distance between them and combining the two closest ones. Once a signature reaches a minimum signature length threshold, it stops participating in this iterative process. Signature merging stops when all signatures are either at this threshold, or merging would result in a trivial signature (e.g. "HTTP 1.1 GET /"). Once the final set of signatures is generated, it is pruned using a set of legitimate HTTP traffic in order to filter out any signatures that match the legitimate traffic. In their experiment, they were able to get 554 signatures from the data, 446 of which they kept.

Their final results for June's data (which contained all of the data they had been collecting) were encouraging, with the final detection rate being 65.1%. Their false positive rate (frequency a signature was incorrectly matched to legitimate internet traffic) was low at .0001% (18 instances) of 12M HTTP queries. However, false positives for these systems can be very hard measure, since the legitimate data is chosen due to its perceived cleanliness. In other words, there can be a great amount of bias if you choose legitimate traffic from a well monitored organization to test the false positive rates for malware signatures that primarily target users behind weak or non-existent internet defenses. Another drawback of their model was noise, which they mention would significantly reduce the quality of the network signatures generated.

Although we had ambitiously attempted to recreate their results after reverse-engineering BadNews and studying the malware samples given to us, it became very clear from human analysis of the data that their method could not be directly applied onto Android malware. There were two main factors that led to our decision to abandon applying their model to Android malware: difficulties in observing malicious behavior from the malware and data noise due to number of legitimate HTTP requests. We inspected the internet data we were collecting by installing and opening the malicious apps, and saw that nearly all the malware included a vast majority of legitimate internet traffic. Furthermore, difficulties in even getting the malicious internet behavior to trigger made it impossible for a fast, automated data-collection system such as the one presented to be used. The authors briefly mention the effect of noise on the model, but the assumption that they make is that the vast majority of the network traces collected are malicious. At the finer levels of clustering, such noise would most likely not be an issue, since only highly similar traces are matched. However, due to cluster and signature merging, legitimate traffic signatures and malicious ones would get merged to form very poor signatures with high false positive rates. We also noticed that their initial coarse-grained clustering would profile the Android malware's HTTP traffic based more on their legitimate or innocuous behavior rather than their malicious behavior. This is a problem since two different malware samples might use the same app for repackaging, which then leads to the malware samples being poorly clustered by the app that the author has hijacked, and not the underlying malicious behavior.

We also took a look at the evasion of such systems, and whether it was possible. Researchers who had developed a system for detecting polymorphic worms in HTTP traffic [3] (which is actually an easier variant of the general malicious traffic problem since most worms use highly specific exploits) were thwarted when Gundy and Vigna managed to beat their system using a self-replicating PHP worm [5]. Their method relied on the insertion of noisy comment blocks, randomization of variable names, trivial instruction insertion and shuffling, and different types of encryption schemes. Since the payload was encrypted, only the decrypting module was subject to detection. However, they were able to bypass even the best signatures that Polygraph generated for the 200 variants of their worm, showing that enough noise can render current web-based detection methods useless once they are taken into account by malware authors.

4 WorseNews

Based on our newfound understanding of malware, we wanted to explore the tools that Android provides malware developers. We created an application that acted as a simple Tic-Tac-Toe game but performed malicious activities in a background service. We used a linux web server as our command-and-control server which hosted MySQL databases and PHP scripts to interact with our application. Once the application is installed on the device, turning on the phone or unlocking the lock screen initiates the malware service. On startup, the

service contacts our non-malicious blogspot and parses the page for a non-visible HTML element that contains the name of our command-and-control server. This system has the advantage of never exposing the name of the real server in the codebase. Once the service parses the blogspot it sends the device id, phone number, and voicemail number to the server. With the READ_PHONE_STATE permission, developers have access to all the meta-information about the device through simple method calls.

After the service posts the device information to the server it proceeds to download an .apk from the same server and store it in the external storage (SD card) of the device if it exists, otherwise saving it to internal storage. Fortunately, developers are not allowed to install any applications without the explicit consent of the user (rooted devices do not have this safety measure). If the download is successful, the user will be prompted to install the application when the phone restarts. The downloaded application contains a sophisticated trojan belonging to the Geinimi malware family. It shares many of the same instructions as BadNews, but it additionally takes all of the user's contacts and sends SMS messages and places phone calls without the user's knowledge. What separates Geinimi from most malware families is that it encrypts the messages exchanged with the server. Without the encryption key, it is impossible to determine what types of information are being exchanged over the network.

In addition to taking the device id, phone number, and voicemail number, the service actively records all incoming SMS messages and incoming and outgoing phone calls. For incoming SMS messages, the phone number, message body, and time are recorded in our database. For incoming and outgoing calls, the number and time are recorded. READ_PHONE_STATE, PROCESS_OUTGOING_CALLS, RECEIVE_SMS, and READ_SMS are the permissions needed to accomplish these actions. Classes can be programmed to respond to any of the events listed above.

We later added an additional functionality to our app whereby a Dalvik executable could be dynamically loaded and run. It was loaded from storage (after being download from our main server) and then using Java's Reflection API we can discover the contents, instantiate classes, and call methods on the data. This allowed us to silently update the Java code in our application, bypassing the Google Play Store's typical update procedures. Having an ability such as this waiting patiently on a smartphone means that a Java exploit tomorrow could be downloaded and then loaded dynamically into an app made today.

5 Evaluation and Results

To prove that our app was a viable candidate as a successful zero-day virus (a virus that goes unnoticed due to its novelty/uniqueness), we installed our app onto a phone along with the top anti-virus software available for download on Android: AVG AntiVirus Free, Lookout, avast! Mobile Security, and Norton Mobile Security. None of the anti-virus systems were able to identify WorseNews as suspicious or malicious. Only two (Norton and Lookout) identified the Gein-

imi sample we download onto the phone, however no link between WorseNews and the malicious .apk was drawn.

Zhou and Xiang note the poor job that Android anti-virus software as a whole is doing to protect against attacks of known samples, showing that in the worst cases only 20% of the malware in their dataset was identified [10]. Given that anti-virus software's sole purpose is for the discovery of previously known malware, it is disappointing to see that even a two year old malware sample can bypass some Android anti-virus software.

The main reason our novel malware cannot be detected is that it does not take advantage of an exploit that need be activated in a very specific manner. Instead, users give us permission to take their information. So from a technological point of view, we are not doing anything that the Android SDK did not intend i.e. it was never really an exploit to begin with from an engineer's perspective.

The Android SDK equips developers with a wide range of tools that make it easy to manipulate the device. While this freedom is typically praised in the mobile development and Android communities alike, one must consider the implications of such freedoms. From a malware developer's perspective, Android provides all of the tools to implement malicious behavior easily. Once a permission has been granted, the developer has full access to that specific functionality on the device. In other words, the permissions system is entirely dichotomous - you are either granted full permission or none at all. Malware authors are only limited by their imagination when it comes to implementation. Given that most individuals do not pay attention to the permissions they grant, malware developers only face one obstacle to infect devices: the Google Play Store [1]. Note: Some countries do not have access to the Google Play Store, and so they use third-party app markets, which are often poorly regulated. These marketplaces are easily penetrated, and can be used for fairly widespread infections.

However, even Google Bouncer (Google's proprietary testing environment to detect malicious apps on the official app market) has come under fire recently for being ineffective at preventing malware from reaching the Play Store. Two web articles [2] [4] recently released suggest that Google Bouncer is not doing enough to prevent the proliferation of malware on the Play Store. They touch upon two major issues with Google Bouncer: the limited time spent with apps and the ability for a developer to fingerprint the Bouncer. The former is obvious, and represents an issue that plagued us as well; malware can often delay its behavior, hence making it more tedious to extract and observe all the possible behaviors dynamically. Google Bouncer supposedly only spends five minutes with each app [2]. The people who discovered this wrote a shell script that executed in the background, profiling the Bouncer session. They also discovered that Google Bouncer could be fingerprinted, which means that if a malware developer knows it is being examined by the Bouncer, it could just deactivate any malicious behavior. Lastly, loading libraries at runtime via the network means that the truly malicious bits of code can wait to hit devices until after release to the public i.e. one waits until after the vetting process to start pushing malicious content.

6 Conclusion

Android malware is no longer the mild annoyance it was once, and should be taken seriously as the threat grows. Despite what many people might think malware is, most Android malware is actually not that technologically sophisticated. Instead, it relies on the ignorance and naivety of the user, who blindly accepts and presses "Yes, Install" to any popup with the words "facebook" and "update" in them. Although anti-virus software is useful for keeping one's phone safe from past threats, the real danger of malware is still present. The current threat of Android malware is that as soon as an undiscovered exploit is uncovered, it can be inserted into the codebase and adversely impact devices immediately.

Android malware is just beginning to take shape, and in the next couple years we predict that more seemingly legitimate apps will turn out to be malware that were dormant for weeks or months at a time. Some believe that BadNews' true value to malware authors was not its SMS fraud capabilities, but rather a proof-of-concept by the malware community as to the possibility of taking advantage of users' naivety and Google's rather permissive platform via the impersonation of a widely used, third-party API (in this case an ad network) [8].

The bad news for Android users is that the BadNews experiment was a huge success. We expect Android malware of the future to exhibit little to no malicious behavior aside from an initialization into the C&C's database and regular pinging for instructions. Once an exploit is released, code to take advantage of it can be dynamically and silently loaded and executed by the malicious app. Most dangerous of all exploits, root exploits, would then provide the C&C with complete control over infected smartphones. A well coordinated attack could easily steal millions of dollars worth of information (credit cards, phone numbers, emails, bank accounts, etc.) within seconds. Detecting these dormant apps would be difficult, especially since their system behavior would appear to be benign.

Although anti-virus companies would be able to respond with signatures for detecting root exploit binaries, there is seemingly nothing stopping malware authors from writing genuinely new Trojan apps, thus bypassing current anti-virus techniques. This just motivates malware authors to be all the more stealthy in their approach, while then executing massive heists of information quickly so as to maximize the benefits prior to getting caught.

However, society is not entirely powerless to the will of malware authors and their financial beneficiaries. More granular permissions could allow for certain domain names and pages to be accessed by certain apps, thus allowing users to more finely control what internet connections are being made. Secondly, with proper education and information, most malware authors would switch to more lucrative careers. By teaching users to be smart about their permissions and knowing when an app is suspiciously overprivileged, the main tactic of the Android malware author is compromised. For instance, one common sense indicator that an app is probably malicious is having lots of fake four and five star reviews. These are usually done by an attacker gaining access to someone's

Google account, and then posting on their behalf. We noticed this potential indicator during our research when one of our phones was infected with a malware masquerading as anti-virus software (common due to the overlap in permissions) on the Google Play Store. We then read the reviews, most of which were clearly stitched together fragments of sentences that had no specific details pertaining to the app (all in broken English of course). It had managed to sneak into the number three spot, all due to a high volume of reviews and downloads. By gaming the review system, it increases the app's visibility thus leading to further infections while also appearing more legitimate.

In fact, education is the only real way to prevent future malware that preys on the ignorant, since it will always try as hard as possible to appear legitimate. Although previous malicious software can have its signatures extracted and utilized by anti-virus software, zero-day attacks are unpreventable. As long as the money is there, the malware developers will be too, and you can be sure they will test each new prototype against the current anti-virus systems to make sure it goes undetected for as long as it needs to.

Although network level detection is possible and perhaps useful in some particular cases, encryption and noisy polymorphism would most likely render it ineffective against stopping all (or enough to be useful) traffic. Larger transfers such as downloads would be encrypted and their keys recovered from publicly accessible blogs and social media. Additionally authors would be constantly changing the schema, form names, encryption algorithms, and encryption keys; all of which could be facilitated using dynamically executed libraries that are remotely downloaded at runtime. Prevention is the only real solution to malware, and detection is merely for the sake of anti-virus companies and the lay Android users who pay them.

References

[1] Serge Egelman Ariel Haney Erika Chin David Wagner Adrienne Porter Felt, Elizabeth Ha. Android permissions: User attention, comprehension, and behavior. *SOUPS '12*, 2012.

[2] Oliva Hou. A look at google bouncer. `http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/`. Accessed: 2013-12-19.

[3] Dawn Song James Newsome, Brad Karp. Polygraph: Automatically generating signature for polymorphic worms. *Security and Privacy, 2005*, pages 226–241, 2005.

[4] Robert Lemos. Researchers beat up google's bouncer. `http://www.darkreading.com/services/researchers-beat-up-googles-bouncer/240002673`. Accessed: 2013-12-19.

[5] Giovanni Vigna Matthew Van Gundy, Davide Balzarotti. Catch me, if

you can: evading network signatures with web-based polymorphic worms. *WOOT '07*, 2007.

[6] Mila Parkour. Contagio malware dump. `http://contagiodump.blogspot.com/`.

[7] Nick Feamster Robert Perdisci, Wenke Lee. Behavioral clustering of http-based malware and signature generation using malicious network traces. *NSDI '10*, pages 25–26, 2010.

[8] Marc Rogers. The bearer of badnews. Technical report, Lookout, 2013.

[9] Lance Whitney. Android snags record 81 percent of smartphone market. `http://news.cnet.com/8301-1035_3-57610229-94/android-snags-record-81-percent-of-smartphone-market/`. Accessed: 2013-12-19.

[10] Xuxian Jiang Yajin Zhou. Dissecting android malware: Characterization and evolution. *SP '12*, pages 95–109, 2012.

Performance Evaluation of Cache Replacement Policies for MiBench Benchmarks

April Bowen Wang
bwang1@swarthmore.edu

Hugh Troeger
htroege1@

Kevin Terusaki
kterusa1@

Abstract

With the increased affordability and power of microprocessors, common appliances are increasingly being embedded with computing power and new ones are being developed. While significant data exists for comparative cache replacement performance for general purpose processors, there has been less research into cache replacement algorithms for embedded systems. In this paper we gather cache miss rates with varying associativity from a large set of cache replacement policies– LRU, PLRU, Random, FIFO (First In First Out), SLRU (Segmented LRU), LIRS (Low Inter-reference Recency Set), and ARC (Adaptive Replacement Cache). We use SimpleScalar, version 3.0, to simulate the cache replacement policies on the MiBench benchmark suite.

1 Introduction

The development of modern processors has increased the potential speed at which computers can accomplish tasks. According to Moore's law, the number of transistors on integrated circuits doubles approximately every two years, which allows more space for transistors on a core. However, the exponential growth in processor transistors does not translate into exponentially greater practical CPU performance because of the existence of bottlenecks such as power consumption and memory latency. As the speed of memory access slows the overall CPU speed, it becomes increasingly important for researchers to investigate methods (Cache Replacement Policies) to reduce the memory reference time.

Caches are employed by CPUs to reduce the average time to access memory. They hold less data, but are quicker to access than main memory. Modern cache structures are often hierarchical, with multiple levels of different sizes, and separate caches for instructions and data. Caches store data that is likely to be accessed again in order to improve performance. Cache replacement policies are algorithms that decide which items will be maintained in the cache, based on which items are determined to be mostly likely to be referenced again.

While significant data exists for comparative cache replacement performance for general purpose processors, there has been less research into cache replacement algorithms for embedded systems. Embedded systems are comprised of microprocessors built in to a mechanical or electrical system with a specialized function. Examples include cars, printers, and microwaves. Common appliances are increasingly being embedded with microprocessors and new ones are being developed.

Some embedded systems deliberately do not utilize caches. Theses systems are those where precision timing and consistency is the paramount concern, and safety is an issue. Caches introduce variability in the time necessary to execute actions, because reading from disk is slower than reading from the cache. However, many embedded systems do not require absolute precision timing in their operations to be useful. With the increase in affordability and

prevalence of high powered microprocessors in everyday appliances, cache performance has become relevant for these embedded systems (Jacob, 1999).

Cache replacement policies decide which blocks of memory to evict from the cache when the cache is full. OPT replacement policy, the optimal replacement policy for any process, would evict the memory block that will be accessed farthest into the future. Since OPT requires a perfect knowledge of future references of a process, it is impossible to implement. Thus, cache replacement policies try to mimic OPT behavior as best as possible to reduce the cache miss rate. The average memory reference time is denoted as

$$T = m \cdot T_m + T_h + E,$$

where m denotes the cache miss rate, T_m = time to make a main memory access when there is a miss, T_h = the time to reference the cache when there is a hit and E accounts for various secondary effects, such as queuing effects in multiprocessor systems. By best approximating OPT, the cache miss rate decreases which effectively decreases the overall memory access time. Given that for any process an OPT policy doesn't exist, modern processors utilize various policies such as Random, Least Recently Used (LRU), Round-Robin, and PLRU (Pseudo LRU).

Though researchers have investigated several novel cache replacement policies and evaluated their performance against the most common policies on general purpose processors, there is insufficient data in comparing performances of different policies under benchmarks relevant to the workloads and instruction patterns representative of embedded systems. In this paper we gather cache miss rates with varying associativity from a large set of cache replacement policies–LRU, PLRU, Random, FIFO (First In First Out), SLRU (Segmented LRU), LIRS (Low Inter-reference Recency Set), and ARC (Adaptive Replacement Cache). We use SimpleScalar, version 3.0, to simulate the cache replacement policies on the MiBench benchmark suite.

2 Related Work

In 2003 Megiddo et al. published a paper describing Adaptive Replacement Cache, which offered several advantages over other algorithms. It outperformed all other online algorithms and came close to the performance of manually tuned offline replacement policies. This paper describes their implementation, which used to create our model of ARC in SimpleScalar. They present data for hit rates and overhead costs comparing ARC to other common replacement algorithms for a variety of cache sizes (Megiddo and Modha, 2003). Our measurements encompass a wider range of cache structures and statistics.

In 2004, Al-Zhoubi et al. discussed the relative performance of many common replacement policies tested using SimpleScalar and the SPEC CPU2000 benchmark suite. They described the algorithms behind LRU, FIFO, random, PLRUt and PLRUm replacement policies. They discuss their relative advantages and disadvantages. They present average hit rates across multiple benchmarks for different algorithms for dL1 caches of different associativities (Al-Zoubi et al., 2004). In addition to the algorithms tested in this paper, we implemented LIRS and ARC using the SimpleScalar framework. We test all these algorithms with the sim-cache tool using the more recent SPEC CPU2006 benchmark suite.

In 2002, Song et. al. developed Low Interreference Recency Set (LIRS) (Jiang and Zhang, 2002) which improved on LRUs performance in cases that exposed LRU's inability to cope with access patterns with weak locality. The authors mention three instances where LRU performs poorly: file scanning, cyclical pattern of accesses that are slightly larger than the cache size, and a multiuser database application that stores index entries and data blocks. In their performance evaluation of LIRS, their tests comprised of the observations listed above. They compare the hit rates of LIRS with common cache replacement policies. Additionally, they look at the performance of LIRS with respects to the size of the LIR and HIR sets. We intend to expand their study of LIRS by gathering statistics on test sets that comprise of programs that don't experience the specific problems the authors observed.

in 1999, Jacob discussed the increasing importance of software implemented cache replacement policies for real time embedded systems. While traditionally many embedded systems do not employ caches because probabilistic speed increases can af-

fect precision and safety, embedded system are seeing a rapid increase in computing power. This leads to higher percentage of these systems having the potential for gains in performance if caches are implemented (Jacob, 1999).

3 Background

3.1 Common CRPs

Common cache replacement policies include LRU (least recently used), FIFO (First In First Out) and Random replacement policy.

3.2 PLRU

The Pseudo-LRU algorithms are devised to model LRU's behavior but are much more feasible than LRU in terms of implementation in hardware. For associativities higher than 4, a PLRU is much more efficient and less expensive than LRU to implement. This algorithm requires less complex update logic and fewer state bits as well. However, it is just an approximation of the LRU algorithm. It is possible for a least recently used block to never get replaced if it is in the same branch as a frequently accessed block. PLRU algorithms are found in the CPU cache of the Intel 486 as well as in several processors in the Power Architecture/PowerPC family.

There are two types of PLRU: Tree-PLRU and Bit-PLRU. The Tree-PLRU is modeled using a binary tree. Each node in the PLRU binary tree has a left child and a right child. When traversing the binary tree, we represent following the edge that leads to the left child as a 0 and following the edge that leads to the right child as a 1. The nodes of the tree store a direction bit that refers to either the left child or the right child and the leaves of the tree store the actual cache blocks. On a cache hit or update, we traverse the tree to find the element we want to access. For each node we visit on the path from the root to the leaf, we update the direction bit stored in the node with the opposite value of the direction we actually follow to access the particular block. On a cache miss, the victim block is found by following the direction bits stored in each of the nodes.

Bit-PLRU is also called MRU based PLRU. Each entry in the cache maintains an MRU bit. These bits are initialized to 0. When the item is referenced the bit is set to 1, indicating that it was referenced recently. When looking for a block to replace, the algorithm selects blocks with MRU bits equal to 0. When there is only one 0 left in the cache, the bits are al flipped. All the ones are set to 0, and the last zero block is replaced and set to a 1.

3.3 SLRU

Segmented LRU combines aspects of LRU and LFU. The cache is divided into a protected and a probationary segment. Elements that are referenced one time are inserted into the probationary segment. When there is a hit on an item in that segment, it is moved to the head of the protected segment. This way item referenced frequently will not be flushed out of the cache by scanning. When an item is evicted from the protected segment it is inserted at the head of the probationary segment. The segment size is fixed.

3.4 ARC

Adaptive Replacement Cache is a cache replacement policy developed at the IBM Almaden Research Center that outperforms LRU (Megiddo and Modha, 2003). ARC combines qualities of the LRU and LFU algorithms in a dynamic system that can adapt to different patterns of data access. It improves on the structure of SLRU to all for online changing of the partition determining the segment sizes. ARC divides the cache into two sections, which hold recently and frequently referenced entries. Ghost list contain entries that have recently been evicted from either cache section. Hits in the ghost lists give information about which type of entry is being referenced more, and the algorithm changes the relative sizes of the frequently and recently referenced portions. This way ARC can adapt to better handle differing program behaviors. The basic structure of ARC is illustrated in Figure 1.

One sections holds the most recently referenced entries (T1), and the other holds frequently referenced entries (T2). When there is a hit on an item in T1 it is moved to T2. There are two ghost lists for entries that have been removed from T1 and T2 (B1, B2). When an entry is evicted from T1 or T2 an entry containing just the metadata is inserted into B1 or B2 respectively. Upon a cache miss, the ghost lists are scanned. A ghost hit will increase the target size of the corresponding cache partition (a hit in B1 will

increase the target size of T1). When entries are inserted into T1 or T2, the partition will adjust so they are closer to their target sizes. The relation of the target sizes to the current sizes determines whether T1 or T2 will evict an entry to make room for a new one. This algorithm is laid out in Figure 14.

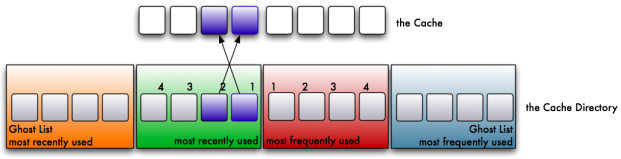

Figure 1: Structure of ARC[1]

3.5 LIRS

LIRS, developed by Song et. al., is another cache replacement policy that improves on LRU (Jiang and Zhang, 2002). LIRS uses IRR (Inter-Reference Recency) as the method for recording history information of each block. The IRR of a block refers to the number of other blocks accessed between the last reference of the block to the current time. The referenced blocks are divided into two sets: a major part containing High Inter-Reference Recency (HIR) blocks and a minor part containing Low Inter-Reference Recency (LIR) blocks. Each block with history information in the cache has the status LIR or HIR. Some HIR blocks may not reside in the cache, but the HIR major part may have entries in the cache that record their status as HIR of non-residence. When a miss occurs, an HIR block from the minor set will be evicted. A block within the LIRS set will always be in the cache and thus will always have a cache hit.

In the event of a cache hit of an HIR block, a new IRR is given to the block which is equal to its recency. Then the HIR block's new IRR gets compared with the recency of the LIR blocks. If the HIR block's IRR is lower than the LIR blocks recency, then the blocks and the HIR/LIR statuses are swapped. The logic behind this is that if an HIR block's IRR is smaller than the LIR blocks recency, then the HIR block's IRR will also be smaller than next IRR of that LIR block. This is because the recency of the next IRR block is a portion of its IRR, and thus can't be greater.

[1] image from http://www.c0t0d0s0.org/archives/5329-Some-insight-into-the-read-cache-of-ZFS-or-The-ARC.html

3.6 SimplesScalar

The SimpleScalar tool set is a system software infrastructure used to build modeling applications for program performance analysis. It can be used to build modeling applications that simulate real programs running on a range of modern processors and systems. Specifically, *sim-cache* provides simulation statistics for all levels of cache (an instruction l1 cache, data l1 cache, a unified l2 cache, etc) with respect to different parameter values for the number of sets, block size, associativity and replacement policy. The original package supports LRU, FIFO and Random, and we implemented six additional cache replacement policies (PLRU_m, PLRU_t, SLRU, ARC and LIRS) to compare their performance.

3.7 MiBench

Mibench, developed by Guthaus et. al, is a benchmark suite targeted towards microprocessors in embedded systems (Guthaus et al., 2001). The suite contains a variety of benchmark categories intended to represent embedded processor functions in industries including–automotive/industrial, office automation, network, and security. Each benchmark contains small and large input files, but we chose to run our experiments only on the small input files. In the automotive/industrial category we measured cache replacement policies on the q-sort and susan benchmarks. The q-sort test uses the well-known quick-sort algorithm to sort an array of strings into increasing order. The small-input data contains a list of words. The susan test is an image-recognition software developed for recognizing corners and edges in Magnetic Resonance images of the brain. The small-input data is an image file of black and white rectangle.

For the office automation category, we tested cache replacement policies on the stringsearch test. Given phrases, the stringsearch test identifies words with a case-insensitive algorithm.

In the network category, we ran cache replacement policies on the dijkstra and patricia tests. Using the input file, the dijkstra test constructs a large graph in an adjacency matrix representation and then performs the well-known Dijkstras algorithm

to compute the shortest path between every pair of nodes. The patricia test consists of a Patricia trie data structure that is often used within routing tables in network applications. The input data is a list of IP traffic from an active web server for a 2 hour period.

In the security section, we evaluated cache replacement policies on the sha test. The sha test is a secure hash algorithm that outputs a 160-bit message digest for a given input. It is used in the well-known MD4 and MD5 hashing functions. It is also a commonly used algorithm in the exchange of cryptographic keys for generating digital signatures. The input data is a small ASCII text file of an article found online.

In order to run the MiBench benchmarks with SimpleScalar, it was necessary to modify gcc to compile for Alpha-OSF based systems. We obtained the source code for gcc, and followed documented instructions to create a cross-compiler to generate the proper binaries.

4 Our Idea

We present a study of comparative cache performance on benchmarks representative of commercial embedded systems. We used benchmarks from the MiBench suite, which includes a variety of programs and functionality typically found in microprocessors in embedded systems.

We modified SimpleScalar's source code to implement the PLRUt, PLRUm, SLRU, LIRS and ARC algorithms. SimpleScalar comes with LRU, FIFO, and random implemented already. With these algorithm all implemented on the same platform we ran extensive tests to gain comparative miss rates across different associativities, for different cache levels.

This paper examines the performance of ARC and LIRS, two of the best cache replacement policies in the context of typical embedded system workloads. We also examine PLRU, a less complex, but very effective and widely used replacement policy. These algorithms have been shown to outperform common cache replacement policies like LRU (Megiddo and Modha, 2003) (Jiang and Zhang, 2002). However these studies have largely been done on general purpose processors. We show how these algorithms compare to the best common algorithms on programs and data access patterns found in embedded systems.

5 Software Implementation

To implement additional cache replacement policies other than the three already provided by SimpleScalar, we modified the cache.c file in the SimpleScalar Simulator code. *cache_create* and *cache_access* are the two functions we modified. In the original cache.c file, blocks were kept in a linked list within each set. Whenever a cache hit or miss happens, respective actions for the specified replacement policy are taken. For instance, if the replacement policy is LRU, the referenced block is moved to the head of the linked list when there is a cache hit. However, for FIFO, the referenced block remains at the same position in the linked list when a cache hit occurs. We adopt the linked list data structure, in our implementations for other cache replacement policies.

For PLRU, we implemented two versions that vary in their approximation of LRU. For PLRU- T, we declared a control array within the set struct that contains the control bits of the binary tree (top to bottom, left to right) and initialized it to be all zero. The size of the control array equals the set associativity minus one. Inside *cache_access*, we wrote helper functions to simulate the cache behavior under PLRU policy in the case of a cache hit or a cache miss.

For PLRU-M, instead of maintaining an array of control bits, each block contains a control bit variable initialized to zero. When a cache miss occurs, PLRU-M chooses a victim block based on the first block it encounters with a control bit of zero. On a cache hit, if there is only one block with a control bit of zero, the algorithm iterates through all blocks and flips the control bits to zero. Then the control bit of the referenced block is set to one.

To implement SLRU, the cache had to be divided into two smaller linked lists. One of the lists contains probationary blocks while the other list contains safe blocks. Additionally, we implemented helper functions that: 1) inserted and removed blocks from their respective lists 2) transferred a block from one list to another. These functions are called during cache hits and cache misses.

We also used these helper functions to implement ARC which shares the core features of SLRU. ARC used the same cache structure of dual linked lists, which we referred to as safe (frequent, or T2) and out (recent, or T1). We further modified the cache struct to include two linked lists of ghost blocks. We created the ghost block struct based on the standard SimpleScalar cache_blk_t struct, but only containing the tag for comparison, and previous and next pointers.

We based the structure of our implementation off of the pseudo code description provided by Megiddo et al. (Megiddo and Modha, 2003) Figure 14. Case 1 is implemented in the cache_hit portion of SimpleScalar's code, with redirection for the safe and out caches. Many parts of our implementation closely mirrored original SimpleScalar functions, with duplication to handle the two cache segments. Cases 2 through 4 were checked for and handled in the cache_miss section.

Replace was implemented as a function, which took the set and chose which block to evict based on the target size. Replace would call the removal function for the chosen cache segment. We made a make_ghost_block function to generate a ghost_blk_t struct with the tag from an input cache_blk_t, whenever a block was evicted from the cache. We built a checker function for keeping the ghost lists limited to the number of block allowed by the ARC algorithm.

Like SLRU and ARC, LIRS divides each set into two lists–the LIR partition and the HIR partition. We represented these two partitions as queues. The HIR partition is only 1% of the total cache size. Since we represented the LIRS cache directory in each set, only one block was allocated towards the HIR partition in each set.

6 Results

We ran six benchmarks (Dijkstra, Patricia, Qsort, Sha, Stringsearch and Susan) of the MiBench set on SimpleScalar. We measured the miss rates of the level-1 data cache as well as the unified level-2 cache with respect to eight different cache replacement policies. The results are shown in appendix A and B. Note that each letter on the right of the figures corresponds to one cache replacement policy (f-FIFO, i-LIRS, l-LRU, m-PLRU_m, r-Random, s-SLRU, t-PLRU_t, a-ARC).

Our results show that out of the three most common replacement policies, LRU performs the best in most cases, whereas Random generally has the highest miss rates. PLRU_t and PLRU_m closely model the behavior of LRU, with PLRU_m performing slightly better than PLRU_t.

Many of the DL1 cache benchmark results reinforce the generally accepted rankings for these algorithms. ARC exhibits lower miss rates than any of the other policies, with LIRS not far behind. Some notable results include DL1 miss rates for the Qsort benchmark, where LRU, ARC, and LIRS perform worse than any of the other policies, see Figure 4. In the L2 tests, LIRS performed particularly well, frequently having the lowest miss rate. This was most pronounced in the run on the Dijkstra benchmark, see Figure 8.

For any given cache replacement policy, the cache miss rates tend to converge after associativity of eight. For some benchmarks, the miss rate of a cache replacement policy reaches its maximum/minimum at a certain number of associativity. This trend is best illustrated in Figure 6 and Figure 7, where the cache miss rate is the highest at associativity of four across all replacement policies.

7 Future Work

Continuing this project we would conduct further analysis on the unexpected results we encountered. We would also implement some of the other best offline cache replacement policies, such as MQ, 2Q and FBR. These poilicies require parameters to be set manually for optimal performance, in contrast to ARC. Because embedded systems often deal with a more limited range of functions, programmer tuned offline policies counld have an advantage over self tuning algorithms with greater overhead. We would need to research what configurations lead the policiy to perform best for a given benchmark. While ARC frequently outperforms these policies (Megiddo and Modha, 2003) on general purpose processors, it would be valuable to gather data on how ARC stacks up against optimally configured offline policies in embedded systems.

8 Conclusion

We have generated statistics for the performance of several cache replacement policies, namely LRU, PLRU, Random, FIFO, SLRU, LIRS, and ARC on the MiBench suite. Our findings largely reflected those of other research on cache replacement policies, but with a few interesting results, namely the poor performance of the top algorithms on Qsort at DL1, and the notably superior performance of LIRS on DL2.

These findings are indicative of the relative effectiveness of these policies for typical embedded system workloads. The gathering of information on this subject is becoming more important as the prevalence of high powered embedded systems increases.

References

Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. 2004. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42Nd Annual Southeast Regional Conference*, ACM-SE 42, pages 267–272, New York, NY, USA. ACM.

D. Grund and J. Reineke. 2008. Estimating the performance of cache replacement policies. In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 101–112.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA. IEEE Computer Society.

Bruce Jacob. 1999. Cache design for embedded real-time systems. In *Proceedings of the Embedded Systems Conference, Summer*.

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA. ACM.

Song Jiang and Xiaodong Zhang. 2002. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 31–42, New York, NY, USA. ACM.

Nimrod Megiddo and Dharmendra Modha. 2003. Arc: A self-tuning, low overhead replacement cache. In *In Proceedings of the 2003 Conference on File and Storage Technologies (FAST)*, pages 115–130.

Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 199–, Washington, DC, USA. IEEE Computer Society.

Jianliang Xu, Qinglong Hu, Wang-Chien Lee, and D.L. Lee. 2004. Performance evaluation of an optimal cache replacement policy for wireless data dissemination. *Knowledge and Data Engineering, IEEE Transactions on*, 16(1):125–139.

APPENDIX A. Dl1 Cache Miss Rates

Dl1 cache size:16kB; block size:16B

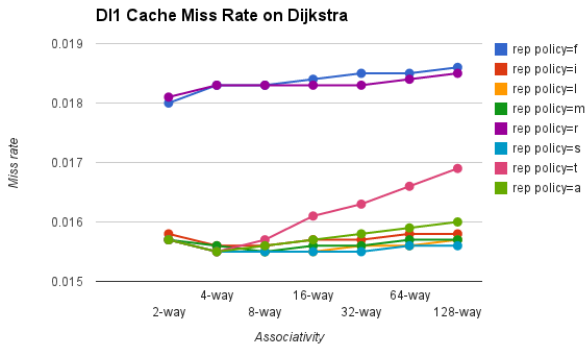

Figure 2: Dl1 Cache Miss Rates on Dijkstra

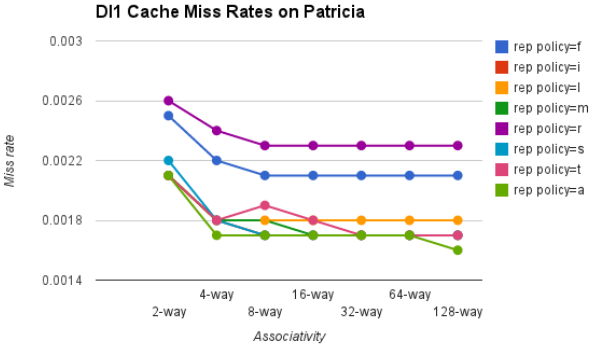

Figure 3: Dl1 Cache Miss Rates on Patricia

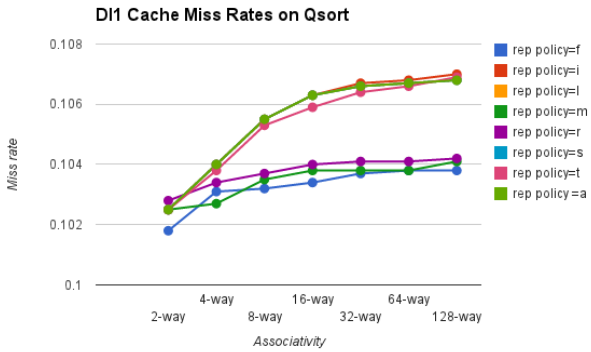

Figure 4: Dl1 Cache Miss Rates on Qsort

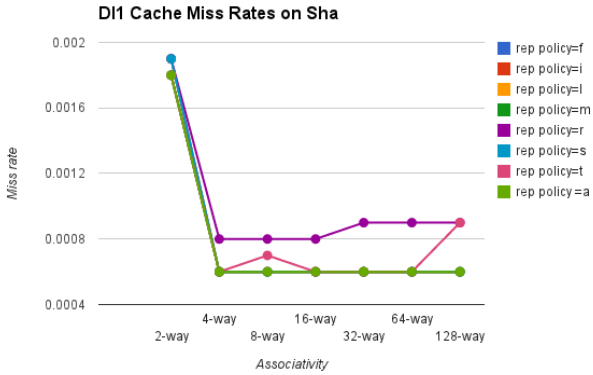

Figure 5: Dl1 Cache Miss Rates on Sha

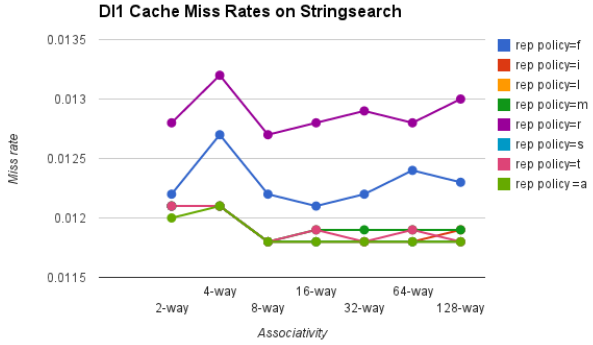

Figure 6: Dl1 Cache Miss Rates on Stringsearch

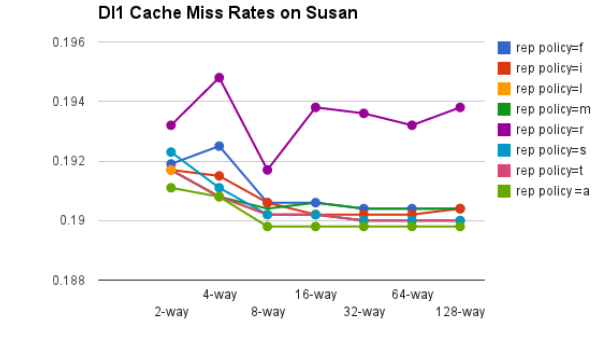

Figure 7: Dl1 Cache Miss Rates on Susan

APPENDIX B. L2 Cache Miss Rates

L2 cache size:32kb; block size:16b; 16kb 4-way dl1 and il1 cache with block size 16b

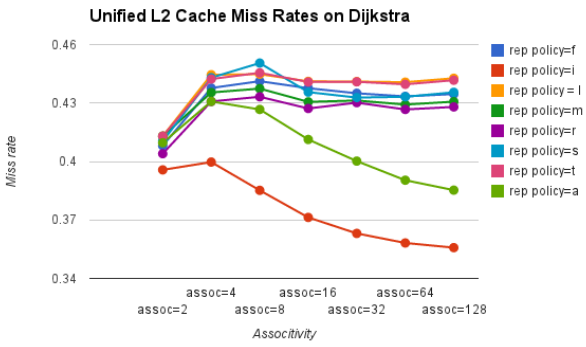

Figure 8: Unified L2 Cache Miss Rates on Dijkstra

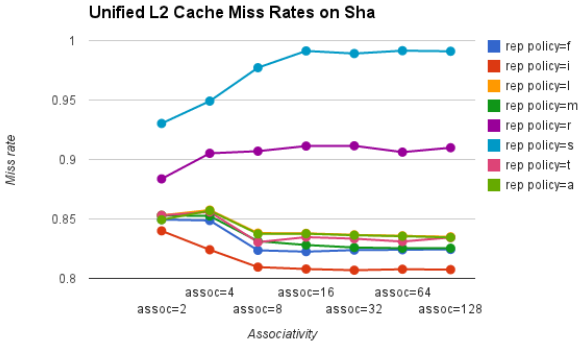

Figure 11: Unified L2 Cache Miss Rates on Sha

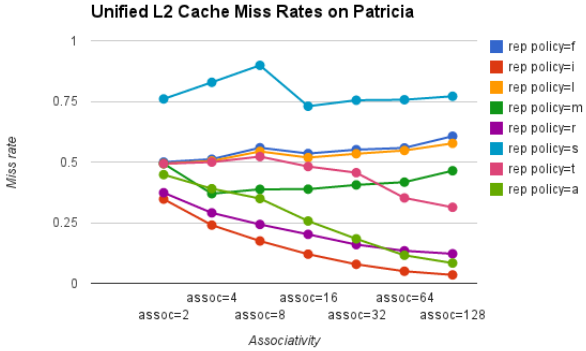

Figure 9: Unified L2 Cache Miss Rates on Patricia

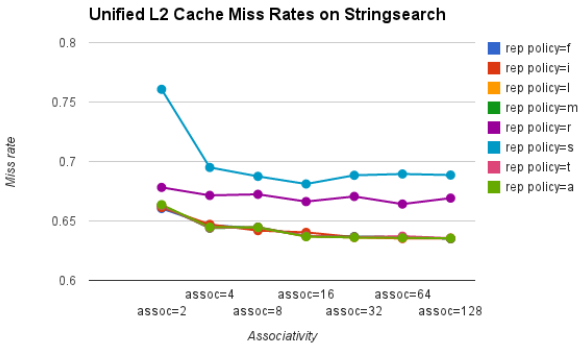

Figure 12: Unified L2 Cache Miss Rates on Stringsearch

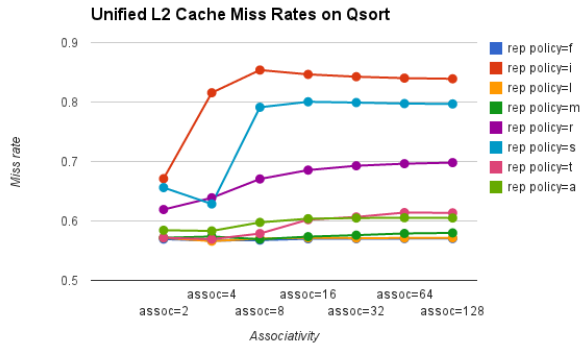

Figure 10: Unified L2 Cache Miss Rates on Qsort

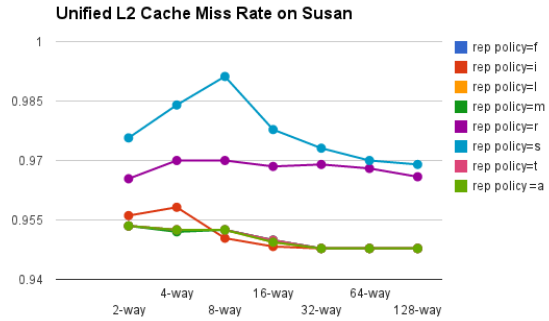

Figure 13: Unified L2 Cache Miss Rates on Susan

Appendix C. Algorithm for Adapative Replacement Cache

ARC(c)

INPUT: The request stream $x_1, x_2, \ldots, x_t, \ldots$.
INITIALIZATION: Set $p = 0$ and set the LRU lists T_1, B_1, T_2, and B_2 to empty.

For every $t \geq 1$ and any x_t, one and only one of the following four cases must occur.

Case I: x_t is in T_1 or T_2. A cache hit has occurred in ARC(c) and DBL($2c$).
Move x_t to MRU position in T_2.

Case II: x_t is in B_1. A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \min\{p + \delta_1, c\}$ where $\delta_1 = \begin{cases} 1 & \text{if } |B_1| \geq |B_2| \\ |B_2|/|B_1| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_1 to the MRU position in T_2 (also fetch x_t to the cache).

Case III: x_t is in B_2. A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \max\{p - \delta_2, 0\}$ where $\delta_2 = \begin{cases} 1 & \text{if } |B_2| \geq |B_1| \\ |B_1|/|B_2| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_2 to the MRU position in T_2 (also fetch x_t to the cache).

Case IV: x_t is not in $T_1 \cup B_1 \cup T_2 \cup B_2$. A cache miss has occurred in ARC(c) and DBL($2c$).

Case A: $L_1 = T_1 \cup B_1$ has exactly c pages.
If ($|T_1| < c$)
Delete LRU page in B_1. REPLACE(x_t, p).
else
Here B_1 is empty. Delete LRU page in T_1 (also remove it from the cache).
endif

Case B: $L_1 = T_1 \cup B_1$ has less than c pages.
If ($|T_1| + |T_2| + |B_1| + |B_2| \geq c$)
Delete LRU page in B_2, if ($|T_1| + |T_2| + |B_1| + |B_2| = 2c$).
REPLACE(x_t, p).
endif

Finally, fetch x_t to the cache and move it to MRU position in T_1.

Subroutine REPLACE(x_t, p)
If (($|T_1|$ is not empty) and (($|T_1|$ exceeds the target p) or (x_t is in B_2 and $|T_1| = p$)))
Delete the LRU page in T_1 (also remove it from the cache), and move it to MRU position in B_1.
else
Delete the LRU page in T_2 (also remove it from the cache), and move it to MRU position in B_2.
endif

Figure 14: ARC Pseudo Code from (Megiddo and Modha, 2003)

Appendix D. Bit and Tree based PLRU

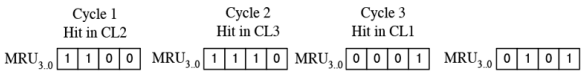

Figure 15: Diagram of PLRUm (Al-Zoubi et al., 2004)

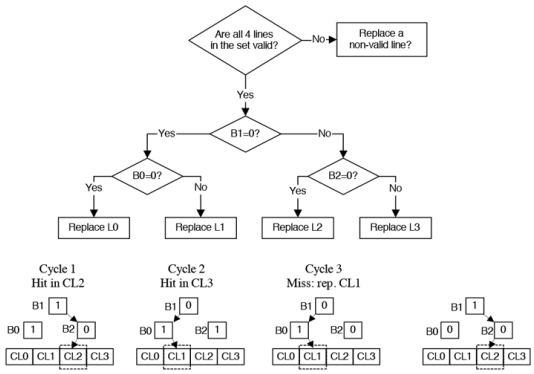

Figure 16: Diagram of PLRUt (Al-Zoubi et al., 2004)

Reverse Engineering NotCompatible

Gabriel Khaselev and Luis Ramirez

December 20, 2013

1 Abstract

The goal of this project is to reverse engineer the prevalent android malware NotCompatible. We obtained samples of NotCompatible from researchers at various universities. The malware was first noticed in the spring of 2012 but made a recurrence recently in 2013. The samples we are trying to reverse engineer are from the second round of malicious activity. By Using static and dynamic analysis to deconstruct the malware we attempted to determine its purpose, method of action, and stealth when present on an infected android device. Based on data examined after reverse engineering NotCompatible we make generalizations about the world of mobile security and in particular the process of understanding android malware and its interaction with command and control servers.

2 Introduction

A wider selection of devices and cheaper entry points contribute to the Android operating system dominating the consumer smartphone market. Smartphones are used as collection bins for personal information, clearly users do not want this information shared over the internet. With 79.3

One event that supports this prediction is the recent resurgence of the android malware NotCompatible. Its first detection was around may 2012 after which there was almost no activity on NotCompatible infected devices. In 2013 this malware once again spread throughout the android community collecting potentially sensitive data on a large number of infected devices. NotCompatible is just one example of the increased number of available android malware on the internet and in circulation. After completely reverse engineering NotCompatible it became clear that it is infact a very sophisticated program, using obfuscation techniques and inherently complex program structure. The complexity of NotCompatible makes it increasingly interesting to examine.

3 Background

NotCompatible is known as Drive-By malware, it infects devices when the user visits a web page that is hosting it or infected.

```
<iframe style="visibility:hidden;
    display:none;"
    src="...websiteURL..."><iframe>
```

Figure 1: An example iframe

Normally, a spam email is opened and inside of the email a link is clicked that redirects the victim to an infected site. Distribution of NotCompatible depends on compromised websites that have a hidden iframe at the bottom of each page such as the iframe shown in Figure **??**. This iframe is present in the fake Fox News website shown in Figure **??**.

If a user visits a compromised website from an Android device, their mobile web browser will automatically begin downloading the NotCompatible application, named SecurityUpdate.apk. The web page informs the user that their current device is not compatible with the site, and that they have the update to fix the problem. If the user has sideloading enabled (an option that allows users to install applications from sources other than the official android market) and they accept the "update", the malware infects their device. Once on the device, NotCompatible may be connecting to private networks by proxying through Android devices connected to the network over wi-fi, this poses an interesting threat and looks much like a botnet.

We obtained two samples of the NotCompatible Malware from Professor Damon Mccoy at George Mason University. One sample comes from the malware's first release around May of 2012, the second is from a resurgence of NotCompatible that occured in the spring of 2013. Both samples seemed very similar if not identical however one was named Update.apk while the other was named SecurityUpdate.apk. The samples connected to different command and control servers. The earlier sample attempts to connect to notcompatibleapp.eu, the domain for which the malware was first named. However, this server is no longer running. Throughout the following sections most procedures were applied to the more recent sample.

4 Reverse Engineering

Neither dynamic or static analysis can be used independently to reverse-engineer a piece of malware. For Android malware, a lack of sufficiently advanced dynamic analysis tools and a large codebase compounds the problem. While static analysis produced the majority of our findings, the dynamic analysis tools that we were able to use provided us clues as to where to focus source code auditing. These two methods in conjuction were effective in the reverse engineering process.

4.1 Tools

There is no standard set of reverse engineering tools for android applications. Through a number of steps we were able to combine several tools namely: Java Debugger, Android Apktool, Dex2Jar, and JDgui. These tools were used to decompile the malware in several steps to obtain resource files, dalvik bytecode, and finally java source code. Lastly we used

Figure 2: The spam page created by NotCompatible.

a dynamic analysis tool called CopperDroid to analyze our sample for network activity, as as a method of running the application in a constant environment.

4.1.1 JDB

Java Debugger was the first tool that we attempted to use on the sample. The process of attaching a JDB session to an android application requires the application to already be running. This prevented us from using JDB to inspect the startup procedure of the malware. Since this included decrypting the data.bin file, we could not observe the unencrypted contents of the file.

Once the debugger was attached, JDB was able to provide a useful thread dump, shown here in Figure **??**. Thread 10 (AsyncTask 1) was of interest, because it was where the network communication most likely resided. Unfortunately, JDB is a very limited tool without access to source code. Setting a breakpoint is entirely guesswork without access to the source code. Even when stopped, JDB does not allow the user to step through instructions at a bytecode level. This prevented us from using the tool further.

Traditional tools used on x86 architectures (GDB, Ida Pro, etc.) do not translate well to the arm architecture or java stack. While it is possible to attach a GDB stub underneath the Dalvik VM, the view provided is too fine-grained to be of real use. While JDB cannot produce enough information, GDB creates too much.

```
Group main:
  (java.lang.Thread)0xc14001f1a8
     <1> main            running
  (java.lang.Thread)0xc1405ac760
     <11> Binder Thread 3 running
  (java.lang.Thread)0xc1405a0670
     <10> AsyncTask 1   cond. waiting
  (android.os.HandlerThread)0xc140591f28
     <9> launcher-loader running
  (java.lang.Thread)0xc14050f208
     <8> Binder Thread 2 running
  (java.lang.Thread)0xc14050f140
     <7> Binder Thread 1 running
```

Figure 3: Thread dump from JDB.

4.1.2 Apktool

Apktool was used to decompile and recompile dalvik vm bytecode. At the bytecode level, we could instrument functions and data structures of interest, and extract the information via a side channel often just writing data to a file. Apktool creates text files in the smali format. It is a relatively new file format used to write dalvik VM bytecode. While access to bytecode was essential in reverse engineering NotCompatible, apktool was particularly useful because it gave us access to the AndroidManifest.xml for the application as well as an encrypted data file. An android manifest details the permissions and intents that the program requires for operation. Throughout the android community there is a stigma against allowing applications with a multitude of permissions, interestingly enough NotCompatible only requires three permissions Internet, Network State,

and Boot Completion. The permissions are seen by the user as detailed in figure 3, notice that there are only three permissions which makes the malware appear less threatening.

Overall apktool was essential in the reverse engineering process because it allowed us to statically modify bytecode to change the malware for testing and provided us with the complete AndroidManifest.xml that details the applications permissions and intent receivers.

4.1.3 Dex2Jar and JDgui

After using Apktool to unpack the .apk and retrieve smali bytecode it became possible to determine what the malware is doing however it is incredibly difficult to follow smali bytecode in a way that sheds light on the programs structure. Thus it became neccesary to convert the bytecode into something more readable, ideally java source code.

Dex2Jar is a decompiling tool used for converting bytecode in the smali format (ending with .dex or .odex) to java class files. This step did not reveal a significant amount of information about NotCompatible but provided the java class files that would later be used as input for JDgui. JDgui is another decompiling tool, it took as input the java class files produced by Dex2Jar and converted them to java source files. It is important to note that these source files are not completely accurate to the java code written by the malware author. Often decompilation causes changes in the source code due to compiler optimizations. One interesting aspect of the source files is the main Run method

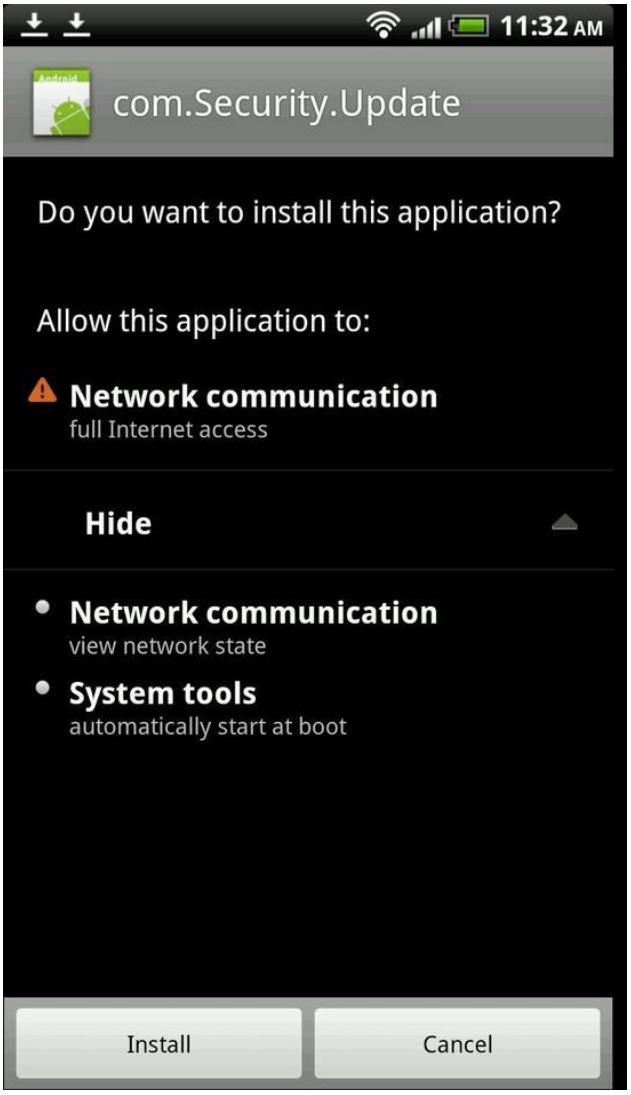

Figure 4: The install dialoug only appears if the users has sideloading available.

of the ThreadServer class (both are discussed in section 5), the Run method could not be decompiled into java instead the decompiler returns an error and prints the function in its original bytecode. It is possible that the Run method was originally written in bytecode by the author of NotCompatible or that some obfuscation techniques were used to prevent decompilation.

At this point we were able to analyze and study the java source files to determine the relationship between several classes in the application. Doing so allowed us to better understand the bytecode with respect to the decompiled java source files we obtained.

4.1.4 CopperDroid

CopperDroid is a tool that automatically performs out-of-the-box dynamic behavioral analysis of Android malware. CopperDroid presents a unified analysis to characterize low-level OS-specific and high-level Android-specific behaviors. Based on the observation that behaviors are enacted through the invocation of system calls, CopperDroid's VMI-based dynamic system call-centric analysis is able to describe the behavior of Android malware. In addition, CopperDroid features a stimulation technique to improve code coverage, aimed at triggerring additional behaviors of interest. CopperDroid uses randomized and targeted stimuli to effectively unveil most aspects of android malware.

We used CopperDroid on a number of occassions, often we would seek to obtain results about network communication NotCompatible is making. Our first run through CopperDroid allowed us to see several important system calls NotCompatible was making. Among these system calls were DNS and HTTP requests to several different servers, most of the requests were communication with NotCompatibles command and control servers. We also noticed that the application was probing mail.live.com and login.live.com, clearly these are proxy connections issued by the command and control server and executed by the infected device. Additionally CopperDroid's results disclosed several file read and write operations on the data.bin file that we obtained using Apktool.

We became curious as to the contents of data.bin. By using the data we received from CopperDroid we were able to determine a method for decrypting NotCompatibles primary data file. Inspection of the source files allowed us to pinpoint the methods used for writing to data.bin, this is show in Figure **??**. We found the appropriate bytecode that executes the save function when writing to data.bin and deleted the encryption, we also deleted the parts of the bytecode that try to decrypt data.bin. Figure **??** shows the portion of the bytecode responsible for encryption, we removed the second and third line and resubmitted the modified sample of NotCompatible to CopperDroid. This technique was fruitful in decrypting data.bin, we discovered the command and control servers are: 244777988244.su and berriesko.ru both using port 443.

```
public void Save()
{
  String str = this.Server1 + "|" + this.Server2 + "|" + this.Port1 + "|" +
      this.Port2;
  try
  {
    DataOutputStream localDataOutputStream = new DataOutputStream(new
        FileOutputStream(this.Owner.getFilesDir().getAbsolutePath() +
        "/data.bin"));
    localDataOutputStream.write(Encrypt(str.getBytes()));
    localDataOutputStream.flush();
    localDataOutputStream.close();
    return;
  }
  catch (IOException localIOException)
  {
    localIOException.printStackTrace();
  }
}
```

Figure 5: Save function. This encrypts and stores the current Command and Control servers to the data.bin file.

```
invoke-virtual {v2}, Ljava/lang/String;->getBytes()[B
move-result-object v3
invoke-virtual {p0, v3}, Lcom/android/fixed/update/Config;->Encrypt([B)[B
move-result-object v3
invoke-virtual {v1, v3}, Ljava/io/DataOutputStream;->write([B)V
```

Figure 6: This is the portion of the bytecode responsible for encrypting and saving the servers to the file.

5 Analysis

Through reverse engineering of NotCompatible we were able to obtain decompiled source code detailing methods, inheritance, and control flow. Figure **??** shows a high level block diagram of the malware. Using an onBootReceiver the malware starts as soon as the device is powered on. The initialization class is called FixedUpdate, it sets up many of the initialization parameters and creates a Config class that handles reading, writing, and encrypting server names and ports in data.bin. FixedUpdate also launches the main ThreadServer whose Run method is used to setup and maintain the communication channel between an infected device and the command and control servers. The ThreadServer starts an instance of NIO Server, the NIO Server class maintains a list o selectors which are used in conjuction with selection keys to connect to server addresses stored in each selector. ThreadServer creates the first instance of a MixerSocket, a class that communicates back and forth with the command and control server using MuxPackets instead of standard network packets. MuxPackets are an obfuscation technique to make packet sniffing more difficult. MixerSockets receive commands such as changes in server names as well as commands to create a proxy connection through the infected device. When the command is sent to establish a proxy connection the address sent from the command and control server to the MixerSocket is used to start a ProxyConnect class. The proxy connect class makes an HTTP request to the address specified by the server. Both proxyconnect and MixerSocket inherit from CustomSocket, a class that implements basic sending and receiving of network packets.

The following is a more detailed analysis of important classes in NotCompatible.

5.1 Config

The config class is instantiated in several places in NotCompatible. Its first instance is created by FixedUpdate. Within config there are plain text strings that define the encyption methods used and even the passkey.

- Ciper: AES ECB NoPadding
- Key Algorithm: AES
- PassKey: ZTY4MGE5YQo

Even with this information it was not trivial to decrypt data.bin, instead we opted to modify the bytecode as mentioned in section 4.2.4. Config is capable of loading data.bin or creating it if it does not already exist and encrypting and decrypting the file during read and write processes. Figure **??** shows the call to load data.bin. All other calls to Config are made in MixerSockets when they receive the command to change the server address or port.

5.2 MixerSocket

MixerSocket is the most important class in NotCompatible because it executes all of the communication that occurs between the servers and an infected device. When the malwarre first starts the MixerSocket establishes a handshake with the command and

control server. The first step is to execute the onConnect method shown in **??**. The input parameter is a selectionkey used to access a particular selector in the NIO Server and connect to the appropriate command and control server. We can see that there is some obfuscation happening, the author of NotCompatible is sending a specific array of bytes in a MuxPacket, the onConnect data being sent from the malware is 040000010500000000007000000. Next the server reads this data, if it is incorrect the connection is immediately terminated. We discovered this through probing the command and control servers in a browser. Finally the server sends back a confirmation in the form of a ping and the MixerSocket responds with a pong as shown in **??**. The array of bytes sent in the pong is 040000010100000005. At this point the MixerSocket can now openly communicate with the server, this means it has two main functions. The first is to change the servers and ports in data.bin by calling config. The second is to pass a parameter sent from the server to a proxy connection to make the HTTP request.

```
public void onConnect(SelectionKey
    paramSelectionKey)
  throws IOException
{
  super.onConnect(paramSelectionKey);
  this.Status = "Connect";
  int i = (byte)(0xFF &
     this.ConnectType);
  int j = (byte)(0xFF &
     this.ConnectType >> 8);
  byte[] arrayOfByte = new byte[5];
  arrayOfByte[1] = 7;
  arrayOfByte[3] = i;
  arrayOfByte[4] = j;
  MuxPacket localMuxPacket = new
     MuxPacket();
  localMuxPacket.dataType = 1;
  localMuxPacket.Data.put(arrayOfByte);
  Send(localMuxPacket.pack());
}
```

Figure 9: onConnect. This function starts the handshake with the server.

```
public void sendPong()
  throws IOException
{
  MuxPacket localMuxPacket = new
     MuxPacket();
  localMuxPacket.dataType = 1;
  localMuxPacket.Data.put((byte)5);
  Send(localMuxPacket.pack());
}
```

Figure 10: SendPong, this function handles the 3rd part of the handshaking procedure.

5.3 proxyConnect

The main purpose of proxyConnect is to send an HTTP request from the infected device to an address specified by the command and control server. As previously stated, the command and control server sends this address to a MixerSocket which passes it to the proxyConnect class. The proxyConnect class then makes the HTTP request and upon receiving

data from the request it forwards this data back to the command and control server by calling its parent MixerSocket and telling it to send MuxPackets containing the new information.

5.4 MuxPacket

The MuxPacket is used when communicating between a MixerSocket and a Command and Control server. The MuxPackets are clearly an obfuscation technique to make it more difficult to unmarshall packets being sent between the malware and the servers. The code below shown in Figure **??** is the decompiled source code for the pack method which scrambles packet data. The use of MuxPackets by NotCompatible makes it significantly more stealthy and much harder to reverse engineer. CopperDroid was unable to determine the contents of packet data sent between the malware and the server, in the results we obtained when running our variant of NotCompatible packet data was incomprehensible.

6 Ramifications and Implications

6.1 Botnets

Moore's law has not overlooked mobile devices. They are quickly becoming almost as powerful as traditional x86 computing systems. Coupled with the fact that mobile devices are ususally run continuously with a constant internet connection, this is an as of yet mostly untapped resource for the electronic crime community. NotCompatible has a plethora of untapped potential, and with the growing power and market penetration of mobile devices, it will certainly not be the last android botnet.

Botnets have traditionally been used to send spam, store stolen information, and DDOS web servers. The author of NotCompatible simply uses the victim's phone as a proxy, presumably to make future attacks more difficult to trace. With the growing popularity of hidden networks such as TOR and I2P, it is not surprising that malware authors might want another method to anonymize themselves. This does not mean that it is incapable of fulfilling more traditional botnet roles. In the current state, NotCompatible could easily be used to DDOS a web service. It can also be used to proxy into private networks that are not normally accessible from the internet. Only slight modifications would be necessary to make it a vehicle for discovering exploits on these and internet-facing hosts. SQL injections, port sniffing, and buffer overflow attacks are all accessible with the current state of mobile platforms. There are numerous other methods that malcontents can monetize malware. For instance, a similar trojan can be used to offload password cracking computations or other highly parallelizable tasks to a large group of compromised mobile devices. Clearly NotCompatible has the potential to be very dangerous.

6.2 Android

Despite how dangerous NotCompatible and other mobile botnets seem to be, the primary focus in mobile security today is in regulating permissions to protect sensitive user data. NotCompatible bypasses this, because it does not target user data. The broader implication however, is that the current system of permissions can be bypassed by ignoring sensitive data on the devices. The permissions system in its current implementation cannot prevent attacks like this from happening. NotCompatible only uses 3 permissions: network state, system tools (to turn on at boot), and internet. Internet permissions is one of the most common on Google Play, and it is not uncommon for an application to ask for 6 or even more permissions. Even to an astute android user, NotCompatible does not seem malicious at the surface level. Although this malware can still incur data and energy costs to the phone users, as well as giving the bot herder a computational leg up.

Another problem with the current android security paradigm is that applications can start a background process without the user's knowledge or permission. NotCompatible does not even have any visible pages. When observed in the runnning processes list, it appears to be a system process.

7 Conclusions

We were able to successfully reverse engineer samples of NotCompatible. The samples we reverse engineered are from the second round of malicious activity. By Using static and dynamic analysis to deconstruct the malware we determined its purpose, method of action, and stealth when present on an infected android device. NotCompatible is a sophisticated mobile botnet, but it is clear that it is simply a proof of concept that such a botnet can be created. In the current state of mobile security malware like NotCompatible is extremely dangerous in that it has many capabilities and is very difficult to detect, especially as the owner of an infectd device. We conclude that the current state of mobile security is ill-equiped to handle the possibility of a powerful android botnet and that in the future steps should be taken to minimize the threat of botnets to the online community and protect ordinary citizens from malicious bot herders.

8 References

- Abu Rajab, Moheeb, et al. "A multifaceted approach to understanding the botnet phenomenon." Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. ACM, 2006.

- Zhou, Yajin, and Xuxian Jiang. "Dissecting android malware: Characterization and evolution." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.

- "NotCompatible Android Trojan: What You Need to Know." TechHive. N.p., n.d. Web. 20 Nov. 2013.

- "'Android/NotCompatible' Looks Like Piece of PC Botnet." McAfee Android-NotCompatible Looks Like Piece of PC Botnet Comments. N.p., n.d. Web. 20 Nov. 2013. ...

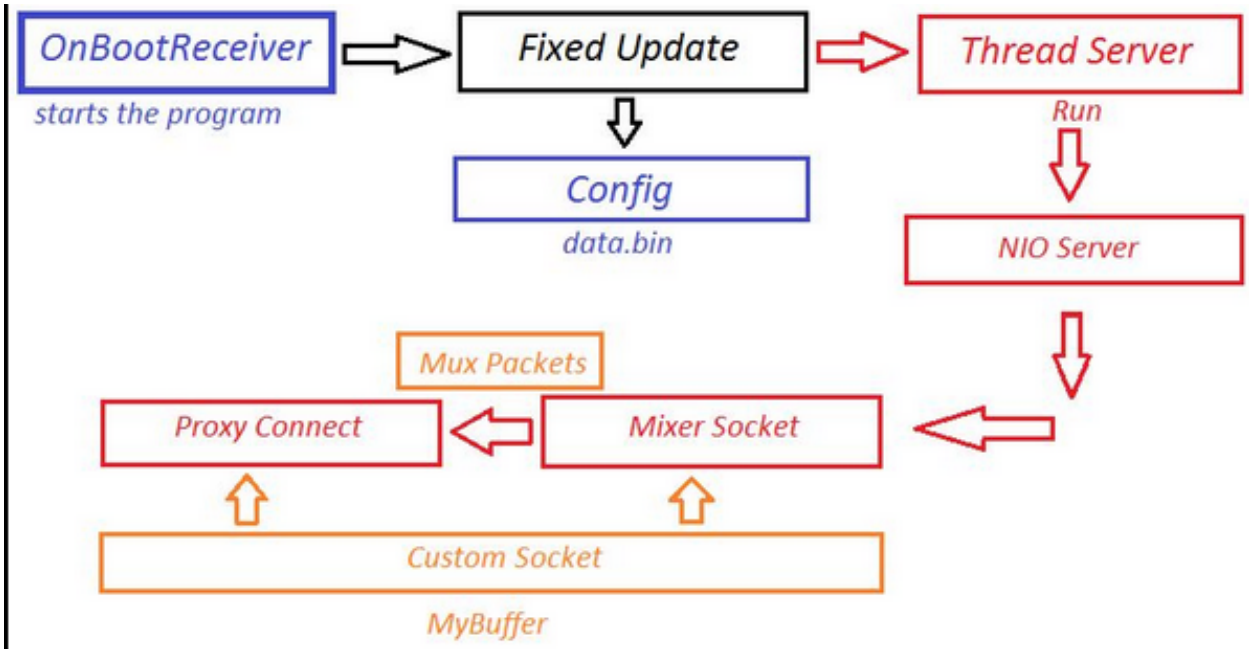

Figure 7: Control flow diagram for Not Compatible.

```
localObject = new FileInputStream(new
    File(this.Owner.getFilesDir().getAbsolutePath(), "/data.bin"));
```

Figure 8: Loading data.bin in Config

```
public MyBuffer pack()
{
  this.length = this.Data.Size;
  MyBuffer localMyBuffer = new MyBuffer();
  localMyBuffer.put(this.version);
  localMyBuffer.put((byte)(0xFF & this.chanal));
  localMyBuffer.put((byte)((0xFF00 & this.chanal) >> 8));
  localMyBuffer.put(this.dataType);
  localMyBuffer.put((byte)(0xFF & this.length));
  localMyBuffer.put((byte)((0xFF00 & this.length) >> 8));
  localMyBuffer.put((byte)((0xFF0000 & this.length) >> 16));
  localMyBuffer.put((byte)((0xFF000000 & this.length) >> 24));
  if (this.length > 0)
    localMyBuffer.put(this.Data.array());
  return localMyBuffer;
}
```

Figure 11: pack. This function scrambles the packets sent in order to provide another layer of obfuscation.