Combining Programs to Enhance Security Software

Yuan Kang

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2018

ABSTRACT

Combining Programs to Enhance Security Software

Yuan Kang

Automatic threats require automatic solutions, which become automatic threats themselves. When software grows in functionality, it grows in complexity, and in the number of bugs. To keep track of and counter all of the possible ways that a malicious party can exploit these bugs, we need security software. Such software helps human developers identify and remove bugs, or system administrators detect attempted attacks. But like any other software, and likely more so, security software itself can have blind spots or flaws. In the best case, it stops working, and becomes ineffective. In the worst case, the security software has privileged access to the system it is supposed to protect, and the attacker can hijack those privileges for its own purposes. So we will need external programs to compensate for their weaknesses. At the same time, we need to minimize the additional attack surface and development time due to creating new solutions. To address both points, this thesis will explore how to combine multiple programs to overcome a number of weaknesses in individual security software: (1) When login authentication and physical protections of a smart phone fail, fake, decoy applications detect unauthorized usage and draw the attacker away from truly sensitive applications; (2) when a fuzzer, an automatic software testing tool, requires a diverse set of initial test inputs, manipulating the tools that a human uses to generate these inputs multiplies the generated inputs; (3) when the software responsible for detecting attacks, known as an intrusion detection system, itself needs protection against attacks, a simplified state machine tracks the software's interac-

tion with the underlying platform, without the complexity and risks of a fully functional intrusion detection system; (4) when intrusion detection systems run on multiple, independent machines, a graph-theoretic framework drives the design for how the machines cooperatively monitor each other, forcing the attacker to not only perform more work, but also do so faster.

Instead of introducing new, stand-alone security software, the above solutions only require a fixed number of new tools that rely on a diverse selection of programs that already exist. Nor do any of the programs, old or new, require additional privileges that the old programs did not have before. In other words, we multiply the power of security software without multiplying their risks.

Contents

Acknowledgements

Learning is about exposing your mind to new ideas, and I have many people to thank for showing me new perspectives throughout my studies.

I am indebted to Vitaly Shmatikov and Ang Cui for giving me my start in cybersecurity research.

Much of my thesis only came out thanks to the help of a number of individuals. I created the mobile decoys during my time at Allure Security, and I would like to thank Erin Ehsani, Joel Peterson, Jon Voris and Shlomo Hershkop for their feedback and support. Suman Jana and Baishakhi Ray have been diligent mentors and collaborators in our work on software security in general, and fuzzing in particular. And I could not have started the work on mutually monitoring monitors had it not been for Simha Sethumadhavan and Kanad Sinha. Moreover, my ability to finish my research is in a large part thanks to the support and patience of Daisy Nguyen, Jorge Espinoza, Derrick Lim, Rob Lane, Dan Cole and Sean Capaloff-Jones. I would also like to thank Boyu Wang for proofreading parts of the dissertation, and giving me feedback while I was preparing for my defense.

My experience during my time at Columbia was made richer due to the people in the department. Steve Bellovin deserves credit for keeping the security group together, and exposing us to perspectives of cybersecurity outside our specializations. Speaking of different perspectives, my knowledge about cryptography is mostly thanks to Tal Malkin, who not only taught me the basics, but also introduced me to research in cryptography. Likewise, I am grateful to Mariana Raykova for the crash course in current areas of research in theoretical cryptography and her contributions to our joint work. Furthermore, I have enjoyed the company of my colleagues, in particular my lab mates, Adrian Tang, Jill Jermyn and Preetam Dutta.

And of course, I would like to thank my advisor, Sal Stolfo, for giving me the opportunity to work in the IDS lab, and at Allure, and for his guidance and encouragement in both my research and life in general.

The work in this dissertation, and my studies are sponsored by Air Force Office of Scientific Research (AFOSR) grant FA9550-12-1-0162 and Office of Naval Research (ONR) grant N00014-17-1-2010. The views and conclusions contained herein are mine and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFOSR or ONR.

To Mom, Dad, William and Boyu.

Introduction

Software is taking on increasing responsibility for real world data and actions. This trend raises two concerns. The increasing responsibility means that an attacker has more opportunities to influence the software to cause it to misbehave. In addition, such misbehavior has a wider impact. Compounding this problem is the increasing complexity of the software to keep up with the responsibilities. So the probability for error in the software also increases, further increasing the opportunity for an attacker to cause harm. Indeed, the last 10 years have demonstrated the extent of harm that insecure software can cause. The Equifax breach shows the risk of buggy software when it handles sensitive information [44]. But the influence of software does not stop at data. The Stuxnet virus targeted industrial control systems, and directly affected the physical world [34].

Attempts to prevent or mitigate such vulnerabilities fall into two broad categories: removing them from the software before their deployment, using a number of software engineering tools and techniques, and detecting and stopping attacks as they occur, using so-called intrusion detection systems (IDS). Seeing how successful attacks are still occurring, we have yet to develop a method that is perfect; not only does each method have its blind spots in finding vulnerabilities or attacks, but its practical use requires automation, which depends on software that is no less vulnerable than those it tries to protect.

While careful setup can isolate such problems when detecting vulnerabilities before deployment, vulnerable intrusion detection systems could be counterproductive. In fact, its responsibility is the security of the entire machine or network, with complexity to match. By the earlier observation of software trends, an IDS is an especially sensitive program.

Compensating for the inadequacies of each protection method requires using multiple, ideally independent, methods, in a framework known as defense in depth, in which the strengths of each method compensates for weaknesses of others [107]. The framework leaves a number of tasks for particular applications that this dissertation will address: picking or designing the set of defenses to complement each other, and making sure that more defenses will not cause more problems. In particular, the new defense should avoid requiring more privileges. Not only would greater privileges make the new system a more useful target for the attacker, but the approach is ultimately unsustainable, since the next defense would have to be even more privileged.

The viability of these criteria forms the hypothesis that the dissertation seeks to verify:

> A defender can compensate for the weakness in an individual piece of security software with a limited, fixed number of new security software, and possibly a boundless number of reused software, none of which require privilege that is unavailable to the original software.

More concretely, the dissertation makes the following contributions:

1. A new line of defense against attackers who have bypassed existing authentication mechanisms on a smart phone, using only privileges available to a normal, installed application.

2. A novel method of automatically generating test inputs by manipulating the inverse of the test program, and multiplying the output a previously manual task.
3. A host-based runtime protection system for intrusion detection systems that avoids the overhead and complexity of an extra, fully-functional detector.
4. A protocol for multiple computers, running independent intrusion detection systems, to protect each other.

The next sections summarize the problems the dissertation tackles, and sketch each solution.

Detecting Attacks against Intrusion Detection Systems

Intrusion detection systems are responsible for the security of a running system, so an attack against them would be particularly damaging. And their complexity is as great as their importance. One common example of an intrusion detection system is anti-virus software. In order to secure an entire system, an anti-virus program needs access to all the data that enters or is on the system. Effectively, the attacker can send input to the software using any method of sending input to the system, even without the user's knowledge [82, 106]. And once the attacker has subverted the IDS, they can abuse its access of the whole system [86]. Besides exposure, the complexity of intrusion detection systems has also been increasing, apparently without raising the complexity of the attacks that they are supposed to prevent. While the size of exploit code has remained constant over time, the number of lines of code in security software has increased by more than an order of magnitude since the 1980s [52]. The increasing complexity of security software is not only due to increasing sophistication, but also a proliferation of formats [54, 106]. It is unrealistic to expect IDS developers to write correct parsers for all formats. After all, the original parser is created by organizations that either designed, or have expertise in the format in question. Yet the the necessity for an anti-virus parser indicates that even expert developers have trouble writing correct parsers. Such difficulty is even more intractable when one considers binary packing formats that malware authors adopt or invent to hide the presence of malicious code [54, 106].

In light of these difficulties, attacks on security software have proliferated. Security researchers have found and published vulnerabilities in anti-virus programs from multiple vendors [82, 106, 54, 81, 37, 13], while vendors themselves have also paid more attention to exploitable bugs in their own software [20]. The appendix lists a sample of recent vulnerabilities discovered in security software. Even worse, malicious attackers have also found anti-virus software to be a valuable target, especially when trying to attack government systems. Initial attacks tried to disable anti-virus software in order to perform other attacks undetected [22]. Since then, vulnerabilities of anti-virus and network monitoring software have been available for sale [81], and attackers have used the anti-virus software as stepping stones to steal government secrets [86, 68]. While the diversity of anti-virus software means that average consumers are unlikely going to be the targets of wide-reaching attacks [54], attackers have already exploited anti-virus software for targeted attacks, as demonstrated in attacks against South Korean targets, which attacked local anti-virus programs [22, 68].

We can dynamically protect intrusion detection systems by using intrusion detection systems for them. The dissertation will propose two approaches. One arranges existing intrusion detection systems to monitor each other, to make sure that all of them are themselves protected, while forcing the user to not only attack them all, but also do so at a speed that increases with the size of the network. This solution takes advantage of the large number and wide distribution of machines, which is particularly prevalent in data centers and sensor networks. Already in 2009, a conservative estimate of the size of cloud providers puts them at a minimum of tens of thousands of machines. And cheap sensor networks are not only deployed in bulk, but also across wide geographic areas, such as on pipelines running over the contiguous United States or Saudi Arabia [74]. The case of sensor networks is particularly relevant not only because of its applicability, but also because individual sensor nodes lack computational power to defend themselves [87], so they will need external protection. The second approach applies to cases where a network of symmetric defense is not possible, usually due to the fact that the monitoring process must be more privileged than the monitored asset. The thesis will argue that more privilege does not necessarily translate into greater complexity. While intrusion detection systems are generally both complex and privileged, the complex portion of the system is actually separate from the privileged part. The intrusion detection system will require a high level of privilege to access potentially malicious data, but the complexity, which is the source of most of the bugs, arises in the analysis part, only after the detector has acquired the suspect data. Forrest *et al.* note that privileged processes in general exhibit regular behavior [36]. Therefore, the dissertation will model this regularity to build a monitoring system specifically tailored for intrusion detection systems, so that while it needs to be able to monitor the original detector, it does not require the full functionality of existing systems; it does not need to access and analyze all of the data of the original detector, but only check if it is not using its privileged functionality in the wrong step.

Improving Fuzzing with Inverse Programs

When detection is unreliable or even dangerous, we can also turn to the preventative side, and reduce the probability that these intrusion detection systems contain bugs in the first place. Researchers who have found vulnerabilities in anti-virus programs recommend this approach, in particular, a technique called fuzzing [54, 108]. In this technique, a program, called a fuzzer, automatically generates inputs to test a program for errors [72]. It is especially well suited for anti-virus programs, where most of the discovered bugs are due to incorrect input parsing, as the vulnerabilities in the appendix show. But while the fuzzer itself is fully automatic, state-of-the-art fuzzers also benefit from pre-selected inputs, or seeds, from which the fuzzers generate further test input [90, 40]. While we know how to select a set of seeds to help the fuzzer test a greater variety of the program's functionality [90], creating them in the first place appears to take manual effort. Creating one or two seeds might require only a basic familiarity with how to use the test program, creating not only a large number, but also a wide variety of seeds, requires greater manual effort and knowledge of non-default usage. It is perhaps inevitable that creating seeds will require some manual effort and domain knowledge, but in many cases, not the entire seed creation process is manual. It generally involves an inverse program whose output is in the format that the test program accepts. The dissertation takes advantage of the inverse program and introduces a technique called inverse fuzzing, which will manipulate the state of the inverse program in order to produce a variety of seeds with only the manual effort required to create one seed.

Defending against Unauthorized Users with Decoy Mobile Applications

While monitoring intrusion detection systems used multiple instances of detectors, and seed generation used the inverse program of the target program, we can leverage arbitrary applications to an old problem in a new setting: authentication on mobile devices. Passwords, or some form of knowledge that the user must remember, is the most popular form of authentication on smart phone devices [43]. But even since its inception, using passwords to authenticate users into computers has been inadequate both in terms of usability and security, a shortcoming that only worsens as users began to use more machines and services that depend on password authentication [75, 4]. And while knowledge-based authentication methods are the most popular, they are still less popular than no authentication at all, which the majority of mobile phone users prefer [43]. The alternative of physical protection also fails, since users lose their devices frequently enough that device loss is one of the top eight threats to mobile computing [21].

Moreover, password authentication suffers from the problem that it cannot authenticate the user after the first login, without degrading user experience. But authentication after login is too great of an opportunity to miss. Protecting the mobile device after login not only follows the principle of defense in depth, but also takes advantage of the availability of data that the user's actions provide [48], and opens the door to a number of user behavior-based intrusion detection techniques.

The particular technique that this dissertation will explore is decoys. Decoys are dummy assets that resemble legitimate ones, but they only serve to entice illegitimate users to access them, and report on the access [7]. For mobile devices, such assets would be applications, through which the user accesses data stored on the phone, or on an online account. We therefore take advantage of assets that we do not even have to create ourselves, in order to catch attackers who have broken through existing authentication methods.

Organization of the Dissertation

The rest of this chapter will discuss related work. Then, the dissertation will start with the defenses tailored to complement specific security mechanisms, mobile authentication and fuzzing, in Chapters 1 and 2, respectively. Next, Chapters 3 and 4 will cover how to defend intrusion detection systems in general: asymetrically on hosts, and symetrically on networks. The last chapter will also evaluate a combination of the host- and network-based monitors.

Related Work

Defending Intrusion Detection Systems

Given the danger of vulnerable intrusion detection systems, researchers who have found bugs in these systems tend to recommend finding and removing bugs before runtime

[54, 106]. If they do propose run-time protection, they see the extra defense as only a stop-gap measure for specific vulnerabilities [73]. That is not to say that designers and vendors of intrusion detection systems have not tried to protect against attacks directed at the detector itself. To understand why this approach is even plausible, it is helpful to classify intrusion detection systems by where they gather their data: from all entry points of interest, or a target process. Anti-virus software and network-based intrusion detection systems inspect all of the data that could affect the processes that they try to protect [85, 93]. On the other hand, a few intrusion detection systems focus on the execution of a target process that generally runs for an extended time. Thus, a monitor that tries to protect an IDS process can be considered a subclass of such process monitors, and it is possible that the process monitor described in the dissertation can apply to less sensitive, long-running processes, such as web servers. Most process monitors focus on what system calls their targets make. System calls are useful data not only because the program cannot obfuscate them, unlike other state that is dependent on the code, such as the instruction pointer value or memory contents, but they are also the interface between the process and the system, and therefore vital if an attack is to have any consequence beyond the logic of the program itself [36]. The benefit of target-focused protection is that it does not have to access and interpret a large variety of input channels and formats, which is the root of most bugs in anti-virus software in the first place. One drawback of target-focused protection is that it will only detect attacks after they have subverted the process. But that is beyond the scope of this work. A second disadvantage is that, in general, focusing on a single process or program does not effectively protect the whole system. But by protecting an existing IDS, we already have a monitor that protects the

whole system, and we simply need to keep it running correctly.

The development of these kinds of intrusion detection systems began with the work of Hofmeyr *et al.* [45], which classified a moving window of system calls as normal or abnormal. This work only considered the identities of the system calls, and not their arguments, and Tan *et al.* were able craft exploits that evaded detection from such simple kinds of system call monitors [98, 99]. Since then, researchers have been trying to consider more information in their system call models. Wagner *et al.* use static analysis to rule out impossible system call sequences [104], while Feng *et al.* examine the call stack [35]. Given the weaknesses of the simple initial efforts, the process monitor described in the dissertation will require more details than only the system call identities. On the other hand, given its sensitive nature, the new process monitor must have a low probability of having a bug, so it cannot process too much of the state of the monitored IDS. The goal is therefore to find a middle ground between the simple and the detailed approaches of previous process monitors.

The following examples of runtime protections for intrusion detection systems all specifically inspect the state of the target process itself. One of the earliest intrusion detection systems, Bro, includes an extra process that detects if Bro is hanging, or has terminated [85]. In industry, anti-virus software also include self-protection mechanisms to prevent attackers from taking them down [92, 97, 51]. The efforts to keep detectors running is closely related to failure detectors in distributed computing, which try to detect hosts that have failed, but can fail themselves [41]. These methods detect if a system has stopped working entirely, and failure detectors are optimized for random, non-adversarial settings.

Other works have considered the possibility that a compromised detector is no longer reliable, even if it is still running. A number of protocols exist to detect compromised hosts. Indra is a distributed protocol in which monitors on hosts detect that a peer has been compromised if they detect an attack from the peer [49], although they do not actively try to determine if a peer is compromised. Yang *et al.* propose a protocol for a distributed system in which multiple hosts can vote on whether a peer is compromised [109]. Compromise of a detector is especially a concern for host-based intrusion detection systems. A host-based intrusion detection system is not trustworthy if the host itself has been compromised. To bypass this problem, Garfinkel *et al.* introduced virtual machine introspection, which puts the host inside a virtual machine, so that an outside monitor has unlimited access to the host's state, and the only way to subvert the outside monitor is when it reads and interprets that state [39]. This idea has spawned a wide area of research, in order to improve performance [89], or ease the interpretation of the virtual machine's state [29]. This thesis will provide a general framework for quantifying the security of systems such as these, and will maximize the security metric while minimizing cost.

By introducing design principles for defense architectures, the work in this dissertation is similar to systems in which multiple monitors protect each other in a cycle [17, 70, 16]. These cycles force the attacker to take down all the monitors, thus maximizing their work. The metric that this thesis will propose will also try to minimize the time that the attacker has, before detection, which is not a trivial matter in networked environments, where detection is not instantaneous.

Fuzzing

Fuzzers are programs that try to find bugs in a test program by automatically generating inputs, running the test program on them, and detecting errors. Thus, they consist of three steps: (1) generating inputs, (2) running the test program, and (3) detecting and classifying errors. The dissertation will propose a method for improving the input generation step in order to exercise more of the test program's functionality.

In order to find bugs more effectively, fuzzers have become more sophisticated ever since purely random fuzzing has popularized the technique [72]. Improvements to fuzzers generally impose at least one of two costs: more domain knowledge, or more complex analysis of the test program. The use of domain knowledge actually predates randomized fuzzing, when the "syntax machine" generated random inputs based on a context free grammar [42]. On the other extreme, symbolic execution tries to calculate inputs to run all execution paths [91, 14, 10, 40]. We can consider seeds to fall into the former category: a human user needs to know how to create inputs for the test program, which is generally a prerequisite for using the program. Seeds have even been so successful that they are even used with fuzzers that use symbolic execution, such as SAGE [40].

The particular fuzzer we will be using is AFL [110], which is a state-of-the-art gray-box, mutational fuzzer. Mutational fuzzers use evolutionary heuristics in order to select the "best" inputs with which to run the test program [15]. That means that they start from seeds, and start mutating them, and use a metric for selecting which generated inputs to further mutate. The metric has generally been code coverage, or the amount of code that the test program has executed [53, 84, 71], although there have been attempts to find alternatives [2, 95]. Such fuzzers are generally gray box, which means that they only analyze the test program in order to determine where to insert code for keeping track of properties including code coverage [8].

While researchers have focused on improving heuristics to increase code coverage and discover more bugs, they have mostly taken for granted the seeding stage. For example, AFLFast, which introduces a number of improvements to AFL, compared the old and new methods using an empty file as the seed [8]. The work by Rebert *et al.* is one of the few that does focus on seed selection, although they outsource the problem of generating the seeds in the first place, by scraping them from the internet before the fuzzing process begins [90]. Inverse fuzzing tries to fill in the gap of generating seeds more efficiently.

Mobile Phone Authentication

Improving Smart Phone Authentication With the shortcomings of password-based authentication, researchers and smart phone producers have been trying to create a more secure and user-friendly alternative. Recent versions of the iPhone let the user log in using physical features of the user, known as biometrics, such as fingerprints and facial appearance [3, 33]. But hackers have already bypassed the fingerprinting method [113], and as of writing time, it is too early to tell if face recognition will be successful or not.

Researchers on the other hand, are more concerned about what happens after the initial authentication, which may or may not be legitimate. Various papers have referred to this field as implicit, active, transparent or continuous authentication, and they all deal with collecting information about the legitimate user's behavior in order to determine if the current activities are legitimate [48, 57, 24]. In some sense, they are an application of anomaly-based intrusion detection systems, which try to model normal, and presumably benign, activities on a system, to classify malicious behavior [26, 55], but on mobile phones, rather than enterprise systems.

Decoys One form of active authentication is decoys. As previously mentioned, decoys are fake assets meant to trick an illegitimate user into accessing it. Their mere existence diverts the attacker from truly valuable assets. But if the defender can detect decoy access, it has proven to be a reliable way of detecting malicious access [7]. Bowen *et al.* were the first to define criteria for effective decoys [9]:

1. Believability: The decoy is indistinguishable from real assets.
2. Enticingness: The decoy asset appears to be valuable enough for the attacker to try to access it.
3. Conspicuousness: The decoy must be accessible to the attacker.
4. Detectability: The defender must be able to detect accesses to decoy documents.
5. Variability: Multiple decoys do not have an easily-identifiable, shared trait. This criterion is related to believability.
6. Non-interference: The decoys do not interfere with the legitimate user's actions. In particular, the legitimate user should not accidentally access them.
7. Differentiability: The legitimate user can identify decoys. This criterion is related to non-interference.

Ben Salem *et al.* introduce another criterion: shelf life. For a decoy to be believable, it should not remain in the system for too long [7].

These properties are not absolute requirements, nor can a decoy satisfy them all at the same time, since some of them oppose each other [31]. For example, a conspicuous decoy is more likely going to interfere with legitimate usage. And a believable decoy might make it harder even for a legitimate user to differentiate from a real file.

Most prior work applied decoys in enterprise or desktop systems, where the most relevant assets are files [7, 101]. The mobile decoys grew out of joint work with Voris *et al.*, which first proposed mobile decoys [102]. Like the previous work on mobile decoys, the decoy assets are applications, rather than files. The new focus is due to the fact that mobile phone users do not access data directly, but through applications. The main difference in the work presented here is that it no longer tries to change applications into decoys, and only uses a central application to mark existing applications as decoys.

The tracking of access to decoy applications resembles parental control applications [28], which attempt to block access to sensitive applications. While the decoy system's self protection mechanism will block some sensitive applications that the user should not access, its primary task is to detect access to applications that are actually expendable, and to report on such actions, and use misdirection to hide information that belongs to the real user.

Chapter 1

Turning Extra Mobile Applications into Decoys

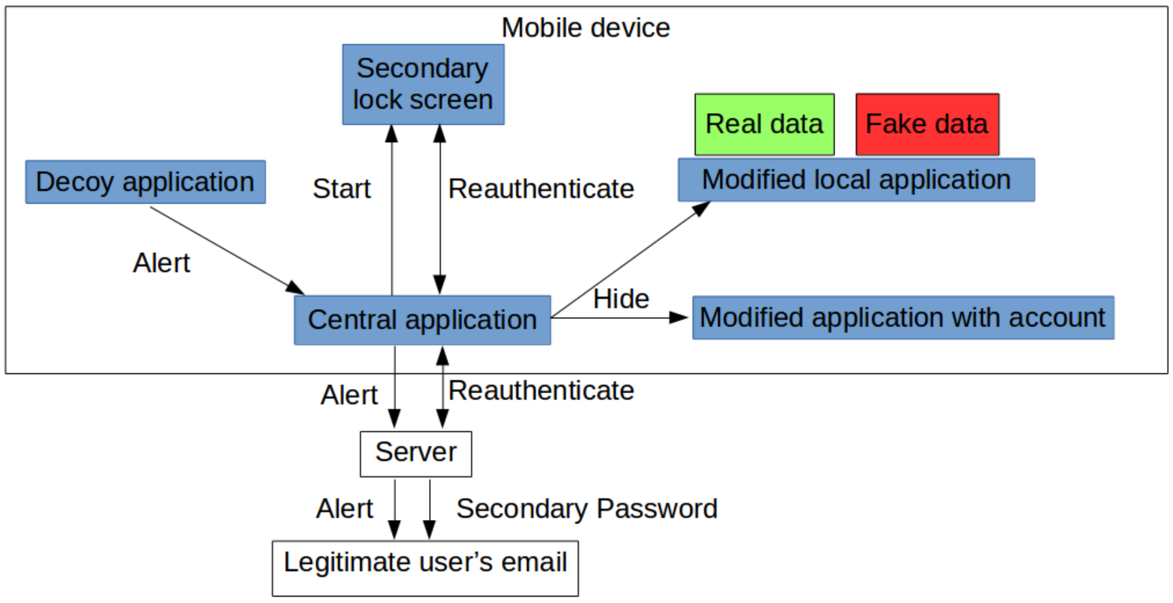
Mobile applications are the gateway to sensitive user data on mobile devices, which they store either on their application-private storage, or on online accounts to which an application is logged in. Accordingly, we will use them in two ways: as decoys that ostensibly contain information that is interesting to the attacker, and as a way of hiding information once the decoys have reported an attack. Since all of the functionality is implemented in applications, the user does not have to make modifications to the phone, aside from installing the components like regular applications, except with some more permissions than usual. In particular, the phone does not need to be "rooted", and users can install the system on phones they receive from their service providers.

The chapter will first describe the threat model, before motivating and outlining the design. The chapter will end with implementation details, including examples of how to modify particular applications to hide data.

Threat Model

The attacker will be a malicious user who has already compromised the existing authentication system. This is a strong form of the device thieves mentioned in the introduction [43]. As a user, rather than malware, the attacker has access to all of the user interface

Figure 1.1: Architecture of mobile decoy system



capabilities, just like the legitimate user, but nothing more. So without further protection mechanisms, the attacker can access applications, manipulate settings, and enable non-administrative shell access via the phone's USB interface [5]. But the attacker must have physical access to the phone, and cannot directly access the private storage of the phone's applications, but must do so through the application.

Architecture

The mobile decoy system consists of the mobile device and an alert server. The former contains decoy applications, an interface for selecting decoys, a central server process for responding to decoy accesses, and modified applications that guard access to sensitive information. The latter is responsible for sending alert data to the user. The components are shown in Figure 1.1

Decoy Applications and Selection Unlike decoy files, decoy applications are not specially created to contain code for reporting access to themselves. Instead, the user chooses real applications to mark as decoys. The user can download them from the application store, or use the ones that already exist on the phone. The disadvantage of this approach is that each decoy application cannot be a self-contained system that can be transferred from one device to another. But this cost is small, since the user cannot store sensitive information in the application program itself. The benefit is that any application can be a decoy, including proprietary applications that resist automated efforts to inject code that would report decoy access. It also lets us take advantage of what is usually considered a problem: bloatware. Bloatware refers to applications that come preloaded with the device. On Android phones, they cannot be removed [69], and attempts to replace them have failed, since the phone refuses to store multiple copies of the same application.

Reporting and Response The decoy system contains a central application that mediates between the decoys, server, and local response mechanisms, which include a secondary lock screen application, and modified sensitive applications that can hide their data.

When it detects a decoy access, it performs the following actions:

1. Start the secondary lock screen application.
2. Send an alert to the server.
3. Lock the phone with the standard lock screen.

In response, the server:

1. Generates a one-time password for the device.

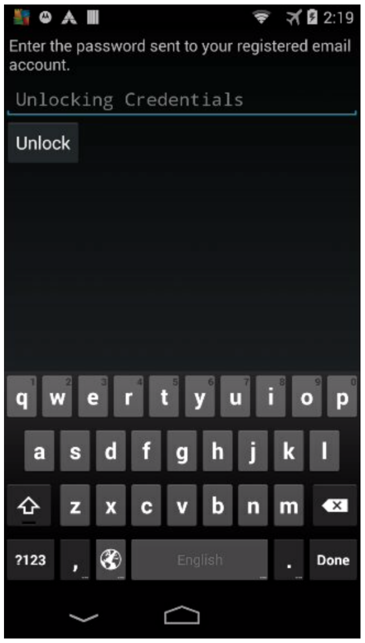
2. Sends an alert, along with the one-time password, to the legitimate user by email.
3. Receives passwords from the secondary lock screen, and replies with an accept or reject message. In case of an accept, the server erases the secondary password.

The secondary lock screen performs multiple functions. The primary function is to serve as an extra layer of defense, in case the attacker has already compromised the primary lock screen credentials. But while forcing the attacker to engage with it, it is also able to record information about the attacker, including audio, visual and location data. The user can also configure the lock screen to serve as a last line of defense, by letting the attacker in, but cause the modified applications to hide the truly sensitive information. When the lock screen appears it:

1. Records a picture with the camera facing the user, sound, and the phone's location, and forwards it to the server, and to the user's email inbox.
2. Presents a password prompt, as shown in Figure 1.2.
3. Upon password entry, relay the password to the server. If the server accepts, close the lock screen. If the server rejects, depending on the user's configuration, either start the lock screen again, or notify the modified application, so that they will hide their data.

Given the sensitive information sent to the server, and the fact that the server is responsible for secondary authentication, the phone performs all interactions with it via TLS, so that eavesdroppers cannot collect information about the user, (who may actually be the legitimate, but clumsy or unlucky), and the attacker cannot spoof acceptance messages from the server.

Figure 1.2: The secondary lock screen


Enter the password sent to your registered email account.

There will be two kinds of modified applications, which require different approaches to hiding data. Traditional applications store their data on the device itself. Thus, erasing them would be counterproductive. Instead, if the data is stored in application-private storage it will access an alternative set of files, which do not contain the sensitive information, and switch back when the user has successfuly reauthenticated at the secondary lock screen [96] (The system does not hide data stored in the general file system since that appears to require changes to the phone). If the data is stored on a remote account, the application logs out the current account, and switches to an alternative, which the user can populate with decoy information. Since the data is stored remotely, there is no need to keep the original information, nor is it safe to do so, since it would increase the number of places the attacker can look for it. Having an alternative account has the additional value that it turns the application into a dispenser of traditional decoy files stored on the cloud.

Satisfiying Decoy Criteria

The mobile application decoys automatically satisfy some criteria, while others depend on choices by the user. Automation of such factors is an area open to future research.

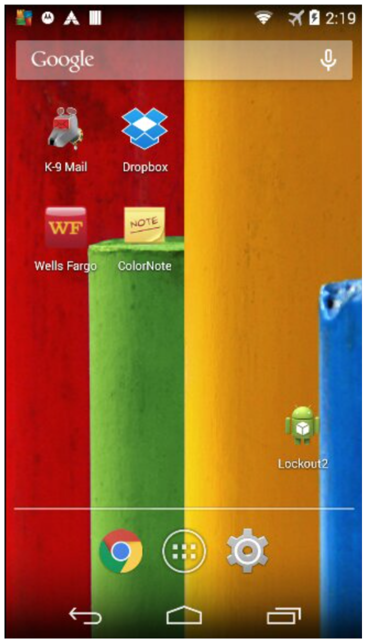
The unconditionally satisfied criteria are:

- Believability: Since all of the decoy applications are real, they are perfectly indistinguishable from non-decoy applications. So in theory, we could further take advantage of believability by deploying multiple copies of the same application, with only one of them being legitimate. This would sacrifice non-interference, and rely on the user's memory to differentiate the real copy, like an extra, application-specific password. But in practice, the difficulty of deploying identical applications prohibits this trick.
- Detectability: The central application can detect any application access. The implementation section will describe the details of how it does so.

Enticingness, conspicuousness, non-interference, and differentiability all depend on the user's choice of applications, and their placement. Figure 1.3 shows a sample use case, where the decoy is a banking application. Moreover, in spite of their poor reputation, some bloatware is likely going to be useful to the legitimate user, since their developers would not push them to the phone if there was no chance that the user would run them. What is unlikely is that a user would use all of them. For example, the author's phone includes multiple different email clients, but he only uses one of them.

The mobile decoys do not meet the last criterion, shelf life. But a long shelf life does not harm believability, because applications are expected to be used for a long time. In

Figure 1.3: A home screen, with a banking application as a decoy, and the other applications in the upper-left corner modified to hide their data. The central application is visible in the lower-right corner, but self-protection mechanisms prevent the user from accessing it.



fact, bloatware can never be removed.

Implementation Details

The mobile decoy system runs on the Android operating system. So it needs to take into account permissions required for particuar actions, possible avenues of subverting or bypassing the decoys system, and the components of the modified applications.

Privileges Used

The implementation of the mobile decoy system uses a number of capabilities that the Android operating system can grant to applications. While some of them require manual configuration by the user, all of them are available to any application, even those that do not come pre-installed with the phone.

To start up along with the phone, the decoy system needs a permission to receive a message from the operating system informing it that the phone has booted [67]. To detect application access, the user needs to set the central application as an accessibility application. This allows it to gather information about the current application in the foreground [100]. By inspecting what application is in the foreground, rather than what applications are running, the decoy system can differentiate between applications that a user intentionally accessed, and applications that automatically run in the background. When the lock screen records pictures, sound and location it needs the respective permissions for those actions [67]. It also needs internet access to contact the server [67]. Finally, locking the phone with the standard lock screen requires the user to configure the central application to be a device administrator, which contains several security-related capabilities that could be useful in an emergency, such as wiping all data on the phone [27].

Self Protection

The attacker can access some applications to render the decoy system useless. On the Android operating system, the Settings application is highly sensitive, but weakly protected. An attacker could use it to uninstall the central application [25], thus disabling the decoy system entirely. Moreover, a user can disable decoy access protection and primary locking by accessing the Settings application, and revoking the disability service and device administrator privileges, respectively [32, 100] [1]. And since the central ap-

[1]In case future versions of Android allow bypassing the Settings application entirely to uninstall an application, a backup application can detect that the central application has been uninstalled [47]. The two applications can protect each other symmetrically using the principles in Chapter 4.

plication itself contains the interface for selecting decoy applications, access to it should also be protected. The latter problem can be solved by programming the central application to require a password chosen by the user. The former problem can be solved by the accessibility functionality itself, by checking if the Settings application is open, and if so, requesting that the user enter the configured password.

Since the secondary lock screen is actually an application window, the attacker could put it in the background by pressing the Home button. To handle such cases, the phone uses a copy of the lock screen, which detects when it is in the foreground, and checks if the secondary lock screen should appear instead. In that case, it reopens the lock screeen.

Offline Fallback

Contacting the server only works if the phone is online. In case the server is offline, alerts must be deferred, and reauthentication requires a second method. When the phone locks while it is offline, the central application will queue the alert data –up to a limited total size – and allow the user to reauthenticate using the password that the user configured for self protection.

Hiding Sensitive Data

If the central application returns access to an unauthenticated user, it broadcasts a message to all modified applications, which change what data they present to the user. Applications that store data in application-private storage will change which files they show to the user. Applications that access remote accounts will log out, and if the central application has dummy account credentials for the application, the modified application will log into the dummy account.

All reverse engineering and modification of proprietary applications was done manually, with the help of APKtool, a program that disassembles and reassembles Android applications [6].

The following examples show that it is possible to change not only open source, but also proprietary applications to hide data.

ColorNote ColorNote is a simple, but proprietary, note-taking application [78]. Reverse-engineering revealed that it stores all data in application-private storage. The modification adds an alternative set of files, and opens from only one set of files, depending on whether the application should show real or dummy files.

K-9 Mail K-9 Mail is an open source mail client [46]. Inspecting the source code revealed a Java class to access some account and mailbox data in private storage, and login methods in a second class that represents a foreground process. Logging out required wiping the private storage information, while logging in required adapting the second class so that K-9 Mail can run the login method in the background, after K-9 Mail has received a command from the central application.

Dropbox Dropbox is a cloud file storage service, with a proprietary mobile application [30]. Nevertheless, modifying the application is almost identical to modifying K-9 Mail, with the exception that variable-name obfuscation made finding the appropriate classes and methods more difficult.

Conclusion

By combining multiple applications, many of which already exist, and some of which require no modification, the mobile phone decoy system provides a second line of defense against failures of the existing authentication system without requiring the user to change the operating system.

Chapter 2

Generating Seeds for Fuzzing Using Inverse Programs

This chapter describes the technique of inverse fuzzing, or how to manipulate inverse programs, in order to automatically multiply the number of seed inputs. It describes prerequisites and limitations for its usability, motivations, architecture and algorithms, integration with a state-of-the art fuzzer, and evaluates its performance. Inverse fuzzing is meant to take the process of generating a single seed, and manipulate it to automatically generate multiple seeds. Thus the evaluation compares its performance to that of the unmodified fuzzer with a manually-generated seed.

Prerequisites and Limitations

Inverse fuzzing can work for any fuzzer that accepts seeds, as long as the tested program has a program, known as the inverse program, that can generate input that the test program accepts. In other words, the inverse program does not need to be a functional inverse in the mathematical sense. But as the experiments will reveal, the technique is not beneficial in every case. Most trivially, while the `cp` program can generate a file of any format –given a proper input file – using it will be no more effective than directly fuzzing its output. Thus, inverse fuzzing is more effective when more of the logic of the inverse program corresponds to the functionality of the tested program. For example, if the tested program checks for a flag bit in a file, the inverse program should have logic for deciding whether or not to set the flag.

While having a small proportion of appropriate logic makes the inverse fuzzer less effective, there is also a feature that unambiguously makes an inverse program useless, or even counterproductive: non-determinism. If non-determinism causes the output to differ even without any manipulation, then it will appear that the inverse fuzzer generated several different seeds, but most of them will not reveal new functionality of the program. Moreover, it will affect the inverse fuzzing algorithm. As we will later see, we will choose mutations adaptively, which is useless if the program's behavior changes between executions with the same choices of mutations. Therefore, we can only use programs as inverse programs if they can produce the desired file deterministically. For example, because `tar` adds a time stamp by default, we had to disable timestamps. On the other hand, ImageMagick's `magick` would only create PNG and PDF files with timestamps, so we could not use them as inverse programs for PNG and PDF parsers. Thus, making inverse fuzzing more applicable requires further work on how to control nondeterministic factors such as timestamps or randomness in cryptographic applications, so that they are both consistent and realistic.

Design

We augment the usual architecture of an existing fuzzer, which runs a test program, with an inverse fuzzer, which runs the instrumented inverse program of the test program.

In more detail, the new components we add are: (1) instrumentations in the inverse program, which can perform mutations on certain values at certain steps, and (2) an inverse fuzzer, which decides what combinations of mutations to perform, and sends seeds to the original fuzzer.

This approach can give us a greater choice of possible mutations than normal fuzzing, but we need to narrow it down to a small set of mutation types. While the original fuzzer influences the test program through the input only, and only uses instrumentation of the test program –if any – in order to gather runtime information, the inverse fuzzer keeps its own input constant, and instead uses the instrumentation in the inverse program to manipulate its state directly. So in principle, the set of manipulations it could perform is a superset of those that the fuzzer can make; the inverse fuzzer could simply fuzz the output buffer, which means that it can at least make the mutations that the original fuzzer makes. However, to optimize effectiveness, we chose specific kinds and combinations of mutations. We will first explain what mutations are, and what types we used, and how to choose them, before explaining how the inverse fuzzer learns about and performs available mutations. At the end of this section, we will describe how the inverse fuzzer and normal fuzzer interact, in order to minimize the performance costs due to the inverse fuzzer.

Mutations

As with regular fuzzers, mutations are a basic building block in generating more inputs, and increasing testing coverage. We want the mutation in the state of the inverse fuzzer to affect the execution of the test program. More specifically, to increase the code coverage of the fuzzer, we would like to change the state of the inverse fuzzer so that the test program executes new code. First, we need to define what a mutation is, and what causes it. Then, we need to know what kind of properties are desirable for a mutation, before choosing what kinds of mutations to use. The rest of the subsection describes the algorithms that automatically enable the inverse fuzzer to make the mutations.

In the context of the inverse fuzzer, mutations apply to the state of the inverse program. A mutation is a single possible change to a value at a single step during the execution of the inverse program. That means that a certain step in the execution can have multiple possible mutations to a particular value. We call such a group of possible mutations a mutation cluster. To make mutations, we add extra code, called instrumentations, to the inverse program. Each instrumentation is responsible for performing one kind of mutation on a single value in the program. When the program executes the instrumentation, it can either preserve the program's functionality, or mutate the value. Since a program can execute the same piece of code multiple times a single instrumentation can be responsible for multiple mutation clusters and mutations. But we need to limit the number of mutations. So for almost all cases, only the first execution of the instrumentation can perform a mutation.

Desired properties of mutations Like most fuzzers, the effectiveness of the inverse fuzzer is related to its code coverage, which measures how much of its own code the test program executes when running all of the inputs that the fuzzer has generated. To increase code coverage, changes to the state of the inverse program should have an analogous change in the test program, and should minimize the effect on the apparent validity of the generated seed file. The former goal is based on the assumption that decisions that the inverse program makes to write certain data to the seed will trigger a corresponding decision in the test program when it tries to parse that data.

The latter goal addresses the common weakness of randomized fuzzers, which is that they are likely to produce obviously invalid input, which the test program will reject right away, before it can execute more interesting code. To reduce the likelihood of this problem, a change in a step of execution in the inverse program should trigger changes in a later step in the execution that is necessary for the seed data to appear consistent. While we could, for example, directly change the seed by changing an output buffer of the inverse program, immediately before it is written to the seed file, such change would not be useful, as the inverse program can no longer compensate for any inconsistencies that the change caused. We therefore want to make sure that the data that the change generated is still available to the inverse program, and that the inverse program had the chance to make further calculations based on it.

To meet the second goal, we will mostly focus on pointers. They include not only buffers, but also any variables, since they are actually a pointer to a memory location, for example the process's stack for local variables, or a preallocated data section for global ones. But, as previously mentioned, some of these buffers might simply be written to the seed, so that changes would make the seed invalid. So we will only consider pointers if their values are used in certain operations.

Types of mutations To meet the first goal of triggering analogous changes in the test program, in particular make it exercise new code, we simply target branches in the inverse program. Branches are instructions that decide what code to execute next. Since branches affect what code to execute next, if we can change the decision that a branch in the test program makes, then we would have made the branch execute new code. Based on the assumed correspondence between the logic of the test program and the inverse program, if we can manipulate a branch in the inverse program so that the output writes a new value, we will change the decision of the corresponding branch in the test program when it reads that value.

The types of branches we will target are if-else branches, and switch statements. If-else branches choose one of two pieces of code based on the truth of a statement. Switch statements consist of a variable and multiple constants, and can choose out of multiple pieces of code, known as cases, depending on which constant the variable matches. In addition, we also treat references in constant lookup tables as switches. Indeed, we have found that compilers often implement switches by creating a lookup table. Since a switch statement consists of only a single variable, that is the one that we will change, to match each one of the constant cases. On the other hand, for if-else branches, generally speaking, the if condition is a boolean, which can be immediately discarded after its use. So if it is practical, it is preferable to mutate the operands in the operation that generates the boolean value. Therefore, we also mutate the operands in comparisons between two values, where the relationship between the operands and output is well-known and simple, so that the output can be easily controlled. If the comparisons are inequalities that decide if one operand is larger or smaller than the other, the mutation can change either operand to cover all three possible relationships between two scalar values: less than, equal to, and greater than. While there are more comparison operations than that, they are all a union of one or more, but not all, of the three aforementioned relationships, so we can get both the true and false results. If the comparison only checks if the two values are equal at all, the mutation will only change an operand so that the two are either equal or not.

To satisfy the second goal, we need to consider how the program will use the mutated values, and perform mutations accordingly. We will focus on bitwise operations, which apply to individual bits of values. [1] If we see a value used in a bitwise operation, then it is likely that the value is used as a bit vector, so we flip the bits individually. Given the abundance of binary operations, however, we limit the values that we mutate. Instead of mutating any pointer that is used in a bitwise operation, we only mutate it if it is a pointer to a buffer or a data structure, which suggests that the value will be passed on, and further used in the program. In particular, a buffer could mean that the change will be written into the seed file (again, only after the changed value has been used in other operations), and a data structure indicates that the values is part of a well-organized state that the inverse program is keeping to translate into a flat output file.

A final type of mutation affects return statements of functions. Upon completion, a function can return a value that the caller will use for further calculation. Some functions may return a value out of a small set of predefined constants (for example a status message, or a type or subtype in programs that support multiple formats or formats with multiple subtypes). But the logic for deciding which constant to return could obscure the fact that the function is only returning one of a small number of pre-defined constants. As a shortcut, we add mutations at return statements, which return some constants that it found to match the type of values that the function could return. For primitive types, such as integers, different functions could assign different meanings to their values, so

[1]We did not consider arithmetic operations, which treat whole values as integers, because they mainly affect integer comparisons, for which we already have mutations.

we only search for constants that appear in that function. For complex data structures, the programmer has assigned a specific use case for the type, so it is more likely that the meaning of the values of a data structure are global (examples include parameters for a specific format or sub-format). In this case, the choice of constants includes all constant data structures of this type that the instrumentation algorithm can find. Moreover, unlike the previous mutations, where each instrumentation be executed multiple times per function call, such a mutation can occur at most once per function call. So we did not limit the number of times an instrumentation can mutate a return value.

Algorithm 1 shows the top-level algorithm for choosing values to instrument from a list of functions. For most interesting statements, it tries to find possible pointer sources of certain variables in statements. In general, for an instruction, i, we represent the operands by $\mathsf{Operands}(i)$. But more specifically, for a switch instruction, $\mathsf{SwitchVar}(i)$ represents the single variable, while $\mathsf{SwitchCases}(i)$ contains all the constant cases that the variable could match. The boolean condition for an if statement is $\mathsf{Condition}(i)$. The algorithm instruments the code with functions shown in Table 2.1. To find the possible pointer sources, Algorithm 1 uses Algorithm 2, which performs a depth first search for pointers that store the value of interest, $p = \mathsf{PointerOfLoad}(v)$, and traverses through instructions that perform some form of copying. One simple class of such instructions is a store to the aforementioned pointer, $\mathsf{Storers}(p)$. Or the value could be a cast of the same value from another type, $\mathsf{CastSrc}(v)$. In addition, the compiler adds so-called phi instructions, which choose one value for a variable as it appears in a particular block in the code, out of different possible values from different branches that converge at that block [105]. We denote the possible sources by $\mathsf{SRC}(v)$.

Algorithm 1 Top level algorithm for choosing values to instrument

```
procedure Instrument(INSTRS)                          ▷ INSTRS is a list of instructions
    for $i \in$ INSTRS do
        if $i$ is a switch or lookup table then
            for $v \in \mathsf{Search}(\mathsf{SwitchVar}(i))$ do
                Insert $\mathsf{MutateSwitch}(v, \mathsf{SwitchCases}(i))$
            end for
        else if $i$ is an if statement then
            $c \leftarrow \mathsf{Condition}(i)$
            if $c$ is not a comparison then
                Insert $\mathsf{MutateIf}(c)$
            end if
        else if $i$ is an inequality ($<, \leq, >, \geq$) then
            $(l, r) \leftarrow \mathsf{Operands}(i)$
            for $v \in \mathsf{Search}(l)$ do
                Insert $\mathsf{MutateCmp}(v, r)$
            end for
            for $v \in \mathsf{Search}(r)$ do
                Insert $\mathsf{MutateCmp}(v, l)$
            end for
        else if $i$ checks for equality ($=, \neq$) then
            $(l, r) \leftarrow \mathsf{Operands}(i)$
            for $v \in \mathsf{Search}(l)$ do
                Insert $\mathsf{MutateEq}(v, r)$
            end for
            for $v \in \mathsf{Search}(r)$ do
                Insert $\mathsf{MutateEq}(v, l)$
            end for
        else if $i$ is a binary operation then
            for $v \in \mathsf{Operands}(i)$ do
                if $v$ points to a buffer element or struct member then
                    Insert $\mathsf{MutateBits}(v)$
                end if
            end for
        else if $i$ is a return instruction then
            $r \leftarrow \mathsf{Operands}(i)$
            if The function's return type is a data structure then
                $K \leftarrow$ All constants in the code of the same type
            else                                  ▷ The function's return type is a primitive
                $K \leftarrow$ All constants in the function of the same type
            end if
            Insert $\mathsf{ReturnSelect}(r, K)$
        end if
    end for
end procedure
```

Algorithm 2 Helper function for searching for sources

```
procedure Search($v$, VISITED $= \emptyset$)   ▷ $v$ is the current value to follow; VISITED is the
list of visited values, and is empty by default.
    if $v \in$ VISITED then
        return $\emptyset$                                ▷ Avoid revisiting the same value
    end if
    VISITED $\leftarrow$ VISITED $\cup \{v\}$
    RESULT $\leftarrow \emptyset$
    if $v$ is a load instruction then
        $p \leftarrow$ PointerOfLoad($v$)
        RESULT $\leftarrow$ RESULT $\cup \{p\}$
        for $s \in$ Storers($p$) do
            RESULT $\leftarrow$ RESULT $\cup$ Search($s$, VISITED)
        end for
    end if
    if $v$ is a phi instruction then
        for $s \in$ SRC($v$) do
            RESULT $\leftarrow$ RESULT $\cup$ Search($s$, VISITED)
        end for
    end if
    if $v$ is a cast then
        RESULT $\leftarrow$ RESULT $\cup$ Search(CastSrc($v$), VISITED)
    end if
    return RESULT
end procedure
```

Table 2.1: Mutation fuctions, and their possible values (including the original)

Mutation Function	Mutated Values	Repeats
$\mathsf{MutateSwitch}(v, \mathsf{SwitchCases}(i))$	$\mathsf{SwitchCases}(i)$	No
$\mathsf{MutateIf}(v)$	$\{\texttt{true}, \texttt{false}\}$	No
$\mathsf{MutateCmp}(v, k)$	$\{k-1, k, k+1\}$	No
$\mathsf{MutateEq}(v, k)$	$\{k, k+1\}$	No
$\mathsf{MutateBits}(v)$	$\{v \oplus 2^j : j \in \lvert v \rvert\}$	No
$\mathsf{ReturnSelect}(v, K)$	K	Yes

Combinations of mutations Given these choices of mutation types, we have several ways of combining them. The only hard restriction is that we cannot make multiple mutations of the same cluster. To narrow down the excess of possibilities, we perform inverse fuzzing adaptively.

We will make mutations in two phases. Initially, we will make only single mutations, and out of the ones that produced new seeds, in the second phase, we recursively try combinations of mutations. In both phases, we will run the test program using a subroutine called RunInverse, which we define as follows: For a set of mutations, MUTATIONS, let RunInverse(MUTATIONS) run the test program using the mutations in MUTATIONS, and if the execution produced a new seed, return the set of new mutations the inverse fuzzer could make, grouped by mutation clusters; otherwise return the empty set. As a result, we will only expand MUTATIONS if it produces new seeds. To be able to keep the old mutations in MUTATIONS, the new mutations that RunInverse can return have two restrictions: 1) they cannot be in the same cluster as any old mutation in MUTATIONS 2) the must occur after the all of the mutations in MUTATIONS. In other words, they can only be applied to values after all execution steps to which any mutation in MUTATIONS applies. The first restriction makes sure that no old mutation is overwritten. The second restriction makes sure that all of the old mutations are still possible, since any mutation will affect the execution of the inverse program after it.

In the first phase, we only perform one mutation at a time, until we have tried each single mutation. Then, out of each one of those mutations that successfully created a new seed, and the new possible mutations after them, we perform the second phase, as follows: Given a set of fixed mutations, MUTATIONS, and a set of possible mutations,

$\mathsf{CANDIDATES}$, for each possible mutation, m, in the second set, $\mathsf{CANDIDATES}$, run the inverse program with the new combination, $\mathsf{MUTATIONS} \cup \{m\}$. If the inverse program produced a new seed program, we recursively repeat the second phase, holding the new combination fixed, and, again, iterating over the possible mutations the most recent mutation we made. Otherwise, we finish the cluster that contains `mutation`, since it appears that mutating the associated value is not producing new seeds.

Algorithm 3 shows the top level of the process, which includes the first phase, and the entry point into the second phase. Algorithm 4 shows the recursive function for the second phase.

While we could have made the first phase recursive, like the second phase, *i.e.* immediately fix the latest, single mutation whenever it produces a new seed, rather than try all single mutations first, we did not do so, because we wanted to start with simple mutations before trying combinations –if the inverse fuzzer even reaches the second phase.

Interaction Between the Inverse Fuzzer and Original Fuzzer

Given the aforementioned choice of mutations, the inverse fuzzer could generate a practically unlimited number of seeds. So we cannot wait for the inverse fuzzer to finish before starting the normal fuzzer. After all, we still would like to use the original fuzzing algorithm, and need to run the test program itself to find any bugs. We would like to run the fuzzer that is more effective at each point in time. But when deciding which fuzzer is more effective, information about past performance will become less relevant. We would also like to only accept generated seeds if they are effective in increasing coverage. This

Algorithm 3 Top level algorithm for iteratively making single mutations, and recursively making combinations of mutations

```
procedure Mutations
    CANDIDATES ← RunInverse($\emptyset$)    ▷ CANDIDATES is a list of possible mutation clusters from when the fuzzer has not made any.
    NEXT_MUTATIONS ← $\emptyset$
    for CLUSTER $\in$ CANDIDATES do
        for $m \in$ CLUSTER do
            MUTATIONS$_m$ ← $\{m\}$
            CANDIDATES$_m$ ← RunInverse(MUTATIONS$_m$)
            if CANDIDATES$_m \neq \emptyset$ then
                NEXT_MUTATIONS ← NEXT_MUTATIONS $\cup$ {(MUTATIONS$_m$, CANDIDATES$_m$)}
            end if
        end for
    end for
    for (MUTATIONS$_m$, CANDIDATES$_m$) $\in$ NEXT_MUTATIONS do
        MutationCombos(CANDIDATES$_m$, MUTATIONS$_m$)    ▷ Using Algorithm 4
    end for
end procedure
```

Algorithm 4 Recursive algorithm for performing multiple mutations

```
procedure MutationCombos(CANDIDATES, MUTATIONS)    ▷ CANDIDATES is a list of possible mutations, grouped by clusters. MUTATIONS is the list of mutations that must be made for all test runs in this stage of the recursion.
    for CLUSTER $\in$ CANDIDATES do
        for $m \in$ CLUSTER do
            MUTATIONS$_m$ ← MUTATIONS $\cup \{m\}$
            CANDIDATES$_m$ ← RunInverse(MUTATIONS$_m$)
            if CANDIDATES$_m \neq \emptyset$ then
                MutationCombos(CANDIDATES$_m$, MUTATIONS$_m$)
            else
                Break    ▷ Break the inner loop, i.e. skip the remaining mutations in this cluster.
            end if
        end for
    end for
end procedure
```

is not as straightforward as accepting only seeds if they increase coverage. After all, code coverage is not the only way of measuring how much functionality the fuzzer has tested, and it could be possible that the fuzzer could mutate two inputs that induce the same coverage, yet produce mutated inputs that exercise different functionalities. Thus, we split inverse fuzzing into the following steps:

1. The inverse fuzzer generates a limited number of mutations. For testing purposes, the limit is 1000 mutations, or 5 seconds, whichever limit the fuzzer reaches sooner. Meanwhile, it calculates the rate of coverage increase over time.
2. The forward fuzzer reads all of the generated seeds, and starts mutating inputs, until it has mutated one input that it has generated itself. Meanwhile, it also calculates the rate of coverage increase over time.
3. Run whichever method has the higher coverage increase. In case of a tie, use the normal fuzzer. For the normal fuzzer:

    - Compare the methods after fuzzing a single input.
    - Calculate the increase of coverage due to fuzzing the input. If the input was generated due by the inverse fuzzer, count it towards the increase in coverage due to fuzzing seeds. Otherwise, count it towards the increase in coverage due to fuzzing generated inputs.

    For the inverse fuzzer:

    - Compare the methods after a time interval, which is 5 seconds for testing purposes.

    - If the inverse fuzzer produces no new inputs at all, disable inverse fuzzing altogether.

   At each comparison, halve the previous coverage and time increases due to the current fuzzer, to decrease the weight of its older performance history, and to give the other fuzzer a chance. If the increase in coverage due to the normal fuzzer fuzzing seeds, divided by the number of fuzzed seeds is no more than the increase in coverage due to the normal fuzzer fuzzing inputs that it has generated on its own, divided by the number of fuzzed, generated inputs, proceed to the next step.

4. This step is the same as the previous step, but the normal fuzzer only accepts any new seeds according to its criteria for accepting inputs that it has generated on its own.

Besides the time the inverse fuzzer takes to generate seeds, and the number of seeds generated, there is a second reason that the inverse fuzzer could actually degrade the performance of the normal fuzzer: the size of the generated input. It is true that both the original fuzzer and the inverse fuzzer can increase the size of an input. But the original fuzzer would have to explicitly populate the extra space, so the increase in size is naturally restrained. On the other hand, the inverse fuzzer could effortlessly grow the seed to an inconvenient size, say by flipping a high bit in a size parameter. Not only do large inputs slow down each execution of the test program, but a thorough fuzzer would have to mutate a large number of input bytes [112]. While large inputs may be interesting for the effect of their sizes on bounds checking and buffer overflows, the values of the individual bits would matter less. Therefore, there is little benefit, and great loss from fuzzing every byte of the input. For large inputs, we therefore wish to only perform length-independent fuzzing. We decide what inputs are large based on how much the original fuzzer could have increased the size of the original seed. More specifically, we keep track of the generation of each fuzzed input, and the maximum increase in size between each generation that the normal fuzzer has ever made. Each generation has a base size, which is the total maximum growth from the original seed. That is, we take the size of the seed generated without any mutations, and add the maximum growth from each generation. Recursively speaking, the base size of a generation is the sum of the base size of the previous generation, and the maximum size increase between the generations. The size limit for each generation is the sum of the base size and the maximum size increase between the generation and the next one. This is effectively a somewhat more permissive limit than what the original fuzzer, with only the unmutated seed, would have ever produced. If the size of an input for a particular generation exceeds the limit of that generation, then the original fuzzer performs only the fuzzing methods whose number of iterations does not depend on the input size.

Implementation

We discuss the implementation decisions we made. In addition to choices of standard tools and operating system interfaces (most of which follow the design of AFL) we also discuss what we used to facilitate interaction between the two fuzzers, and practical reliability concerns.

Implementation Tools and Techniques

The Original Fuzzer For our original fuzzer, we take AFL, version 2.49b [111], and modify it to interact with our inverse fuzzer.

Instrumentation Similar to AFL, we instrument the inverse program using an LLVM pass [114], and write wrappers around clang, to build the inverse program with the pass, and link it to an object file containing the functions that perform the mutation [111]. This limits the extent of Algorithm 1 to a single module, which is all the code (usually a C or C++ source file, and header files) that goes into creating an object file. On one hand, this limits how far back we can search for values to instrument. On the other hand, it avoids slowing down the compilation process too much.

Interprocess Communication Again, like AFL, we use shared memory [111], except instead of simply using it to detect the execution of basic blocks in the test program, we use the shared memory for two-way communication. One side effect is that if the inverse program actually consists of a sequence of multiple programs, then we can communicate with all processes without any extra effort. In practice, we could construct more complex inverse programs out of scripts that run instrumented binaries. But since that involves extra domain knowledge on the part of the user, we have not chosen to use them for comparison with manually-seeded fuzzing.

The one issue that shared memory raises is that we have to preallocate it. The main consequence is that we would have to allocate enough space to include all the mutations we wish to make. While we could reallocate the shared memory for each execution, we simply allocate an estimated upper limit: After running the inverse program for the first time, without any mutations, we allocate enough space to include all the possible mutation clusters. We would only lose potential mutations in the unlikely case that we have a combination of mutations that exceeds the total number of possible clusters, including ones that do not generate new seeds.

Representing Sets of Possible Mutations We will represent mutation clusters and mutations as numbers. Most crucially, we will number mutation clusters by order of appearance. Thus the list of possible mutations only needs to be represented by one number and a boolean. The number represents the first available mutation cluster. The instrumented program also returns a boolean that tells the inverse fuzzer if the last mutation is the last one in its cluster. So when the inverse fuzzer wants to change the last mutation, it knows whether it should use the same cluster, with the next mutation, or start with the next cluster.

Synchronizing the Inverse and Original Fuzzers We use a file lock [58] so that only the inverse fuzzer, or the original fuzzer runs at one time. This ensures that the original fuzzer doesn't run until the inverse fuzzer has created enough seed inputs, the original fuzzer isn't reading a file while the inverse fuzzer is writing to it, and we only run the inverse fuzzer when the original fuzzer needs it to. While a more fine grained use of the lock can satisfy the first two conditions, while also letting both fuzzers run concurrently, we did not do so, because during evaluation, such a design would give our fuzzer more threads than the original fuzzer.

This synchronization method does not prevent the inverse fuzzer from generating seeds once the original fuzzer has terminated. But since the inverse fuzzer runs for the sake of the original fuzzer, we also expect the inverse fuzzer to terminate. So the lock file also contains the process ID (PID) of the normal fuzzer, as long as the normal fuzzer may still need the inverse fuzzer to run. If the inverse fuzzer determines that the lock contains no PID, or there is no longer a process with that PID, it will terminate.

Practical Issues

Ideally, a fuzzer reruns the inverse or test program multiple times, independently. In reality, the test program can affect the system in a way that not only affects the testing result, but could also make the operating system unusable. A number of inverse and test programs affect the file system during the fuzzing process, either because they output directories and files, or because they create temporary ones. This problem is true for all fuzzers, but especially acute for our fuzzer, because we create more seeds.

One problem occurs mainly with archiving programs: The inverse or test program could change permissions in the file system, which could interfere with the fuzzer. For example, by default, `tar` could change the permissions of the current working directory. The fuzzer would run the test program in the fuzzer's output directory, so not only would the test program no longer run properly, but the fuzzer itself would also be unable to continue at all. To prevent such permission changes, we run all of the tests with unprivileged users, who do not own the folders in which they run.

The more common problem is disk usage. Many of our test programs produce output files, and some of them generate temporary files. A properly working program should generally erase temporary files before it terminates. But sometimes, the fuzzer could decide that it has been running for too long, and kill the test process, so that the temporary files start to accumulate. To avoid excessive disk usage, we developed two cleanup routines. The first solution is specific to our modification of the original fuzzer. We generally ran the test program to output any files in the output directory of the fuzzer. If we couldn't, the test program was still restricted to that directory, and the temporary folder, due to the user's permissions (the hosts were dedicated to running the fuzzer, so there were no other users who could have created world-writable directories). At the same time, the fuzzer also uses files in that directory to keep track of its state. So our cleanup routine, between a number of test program executions, would delete any files in the working directory that are unrelated to the fuzzer's state. As we will see, our experiments compared the performance of our workflow against the baseline performance of an unmodified fuzzer, using manually-generated seeds. So we could have applied the same change to the baseline, but we did not. Since we did not want to make modifications to interfere with the baseline performance, and also because the unmodified fuzzer never produced too many inputs that would cause the test program to output files, we did not add this modification to the baseline fuzzer. On the other hand, the temporary files did prove to be a problem in both cases, and previous AFL users have also observed this problem [88]. To mitigate this, we developed an independent script that would clean up old files in the temporary file directory. As we noted previously, this was the only other folder the user could write to. We would run this script throughout our experiments, for both our workflow, and the unmodified fuzzer.

Experimental Evaluation

We compared our fuzzer combination with AFL with a manually generated seed. The section describes this setup in more detail, and shows quantitative results, as well as bugs that we have found with the inverse fuzzer.

Setup

To compare our fuzzer combination with an existing fuzzer that uses manual seeds, we considered the following two methods: 1) Our inverse fuzzer, with an instance of AFL 2.49b, which we modified to interact with our fuzzer, and 2) an unmodified instance of AFL 2.49b. All tests ran for 12 hours. Our fuzzing tests ran on VMWare ESXi instances on Dell PowerEdge R710 hosts, where each virtual machine has 8 2.67GHz processors, and 16GB of RAM. For compilation, we use a separate machine with the same RAM size and processor speed, except the machine only has 4 processors, out of which we use 2 when compiling. That is because we only needed to run each compilation once, so we did not need a high level of parallelization.

Next, we discuss some additional tools that we needed to write for the sake of our experiments.

Coverage Calculation We use the LLVM toolchain for calculating the coverage. It requires that we build the test program with additional flags [94], which we include in a wrapper to `clang`, just like when we instrument the test and inverse programs. An additional challenge arises when we want to gather coverage information over time. It is too inefficient to calculate coverage as the fuzzers are running. But we cannot depend on timestamps to determine when the input files were generated, because AFL could trim them later [112]. We therefore copy each version of an input file as it is generated, which we can detect using inotify [66]. The copy will contain the correct timestamp, which we use for determining when the coverage increased due to the file.

Termination To make sure that both methods are running for the same amount of time, we also need to automatically stop AFL after a predetermined time. But AFL itself does not have an automatic method for stopping; it requires the user to send it the interrupt signal [112]. So we run both versions of AFL with a parent program that sends the fuzzer the interrupt signal after a specified amount of time.

Quantitative Results

We evaluate the inverse fuzzer using the Binutils programs [2], which Böhme *et al.* use to evaluate AFLFast [8], and most of the programs listed by Rebert *et al.* [90], except those that are GUI programs, which AFL does not support. Those programs include `ffmpeg` [3], `gif2png` [4], `jpegtran` [5], `magick` [6], `mp3gain` [7], `mplayer` [8], `mupdf` [9] and `pdf2svg` [10]. For generating sound files, we used `lame` [11] as the inverse program. We also tested archiv-

[2]`http://ftp.gnu.org/gnu/binutils`

[3]`http://ffmpeg.org/releases`

[4]`http://www.catb.org/~esr/gif2png/`

[5]`http://ijg.org/files`

[6]`https://www.imagemagick.org/download/releases/`

[7]`https://packages.ubuntu.com/trusty/mp3gain`

[8]`http://www.mplayerhq.hu/MPlayer/releases/`

[9]`https://mupdf.com/downloads/archive/`

[10]`https://github.com/dawbarton/pdf2svg`

[11]`https://packages.ubuntu.com/xenial/libmp3lame-dev`

Table 2.2: Projects and build times, in seconds, for AFL fuzzer, and inverse fuzzer. Not every program is fuzzed or inverse fuzzed.

Program(s)	Version	AFL	inverse fuzzer
binutils programs	2.28	144.38	146.82
`clamscan`	0.99.2	127.80	
`cpio`	2.12	108.01	108.03
`ffmpeg`	3.3.2	487.97	
`gif2png`	2.5.11	0.84	0.70
`jpegtran`	9b	16.53	
`lame`	3.99.5		26.75
`magick`	7.0.6-10	269.41	227.79
`mp3gain`	1.5.2-r2	2.68	
`mplayer`	1.3.0	465.47	
`mupdf`	1.11	125.88	96.40
`pdf2svg`	0.2.3	1.94	
`tar`	1.29	79.37	76.71

ing programs, `cpio` [12] and `tar` [13]. Moreover, we used AFL to fuzz a modified version of `clamscan` from the ClamAV anti-virus project [14] using some seeds generated by the inverse fuzzers to look for bugs, and to train the monitor in the next chapter. Table 2.2 shows the build times for the projects for AFL and the inverse fuzzer. The extra build time for the inverse fuzzer is at most 5 minutes, which means that the instrumentation for the inverse fuzzer does not make using an inverse program burdensome. Table 2.3 shows the coverage and number of crashes found by the original setting and the inverse fuzzer.

Images 2.1 to 2.22 show the coverage for both fuzzers over time.

As we can see both from the coverage and number of crashes found, the inverse fuzzer beats the normal fuzzer in a vast majority of test cases. The exceptions are programs that parse images or files that can be converted to and from images (PDFs) [15]. Another excep-

[12]`http://ftp.gnu.org/gnu/cpio`

[13]`http://ftp.gnu.org/gnu/tar`

[14]`https://www.clamav.net/downloads`

[15]`magick` lets us create PDFs out of images, but we did not use it as an inverse fuzzer, because it inserted

Table 2.3: Fuzzer performance comparisons. The better results are underlined.

		Coverage		Crashes	
Test program	Inverse Program	Normal	Inverted	Normal	Inverted
c++filt	nm	0.05660	0.05851	0.0833	0.25
magick GIF to JPG	magick PNG to GIF	0.04577	0.05281	0	0
magick GIF to PNG	magick PNG to GIF	0.05367	0.04722	0	0
cpio	cpio	0.2244	0.2646	4.75	39.58
ffmpeg	lame	0.05810	0.06406	0	0
gif2png	magick from PNG to GIF	0.7429	0.7443	9.75	9
jpegtran	magick from PNG to JPG	0.1830	0.1952	0	0
mp3gain	lame	0.5001	0.5166	67	106
mplayer	lame	0.05755	0.06028	0	0
mupdf	mupdf with PDF from picture	0.06013	0.06011	178.5	60.75
mupdf	mupdf with PDF from text	0.1174	0.1181	0	0
nm	as	0.09902	0.1021	0	0
nm	objcopy	0.1042	0.1285	0	0
objdump	as	0.06406	0.07143	0	0
objdump	objcopy	0.06846	0.08885	0	0
pdf2svg	mupdf	0.3281	0.3281	0	0.67
readelf	as	0.1752	0.1917	0	0
readelf	objcopy	0.1776	0.1826	0	0
size	as	0.06135	0.06146	0	0
size	objcopy	0.06220	0.08117	0	0
strings	objcopy	0.002234	0.002234	0	0
tar	tar	0.09273	0.1249	0	0

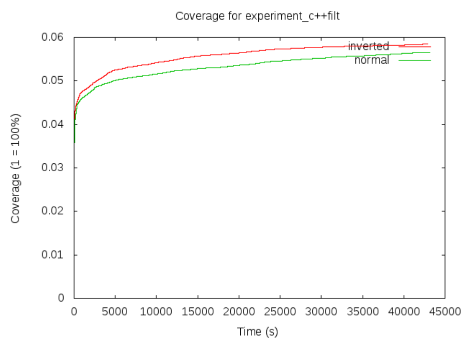


Figure 2.1: Coverage for `c++filt`

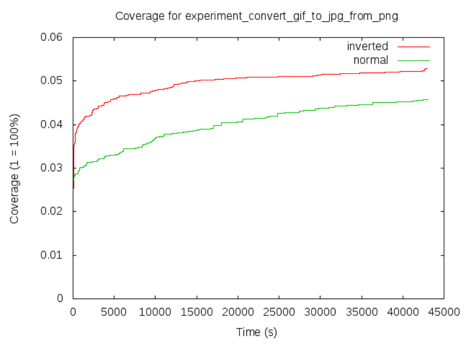


Figure 2.2: Coverage for `magick` converting GIF to JPG from `magick` generating GIF from PNG

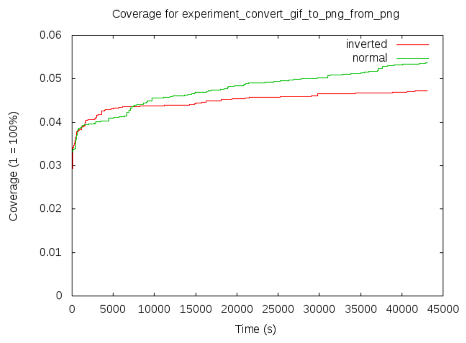


Figure 2.3: Coverage for `magick` converting GIF to PNG from `magick` generating GIF from PNG

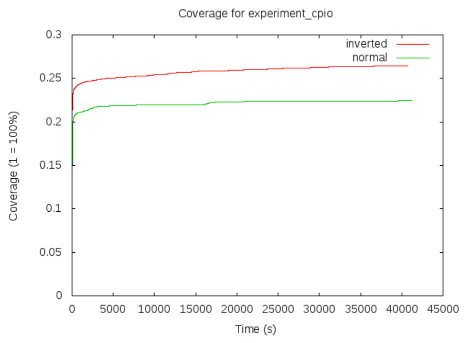


Figure 2.4: Coverage for `cpio`

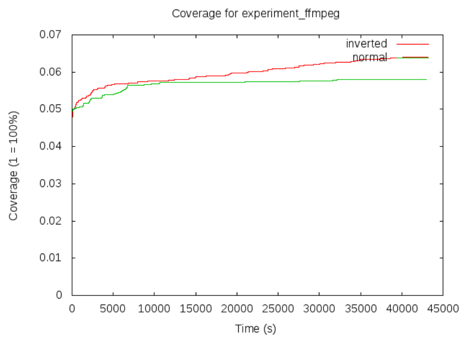


Figure 2.5: Coverage for `ffmpeg`

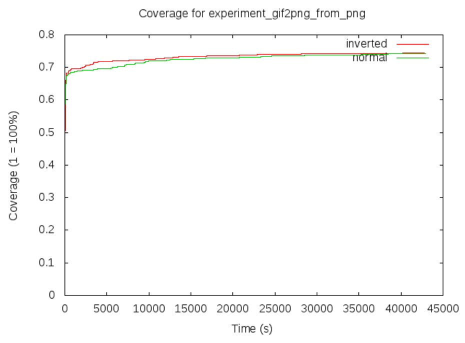


Figure 2.6: Coverage for `gif2png`

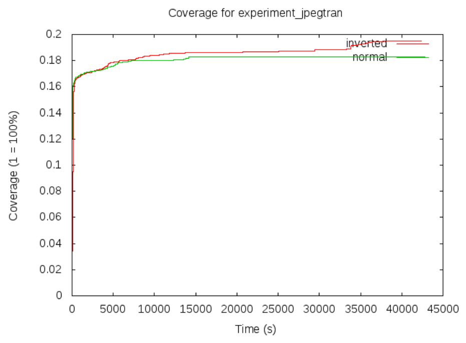


Figure 2.7: Coverage for `jpegtran`

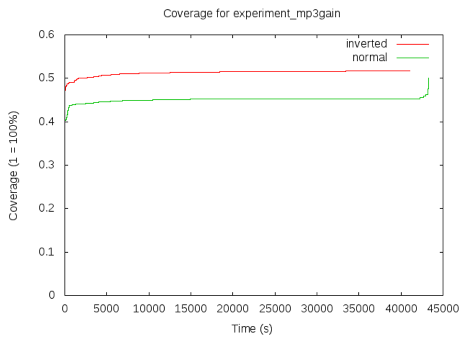


Figure 2.8: Coverage for `mp3gain`. In spite of the spike near the end for the normal fuzzer, its coverage does not beat the 12-hour coverage of the inverse fuzzer even if we let it run for 24 hours.

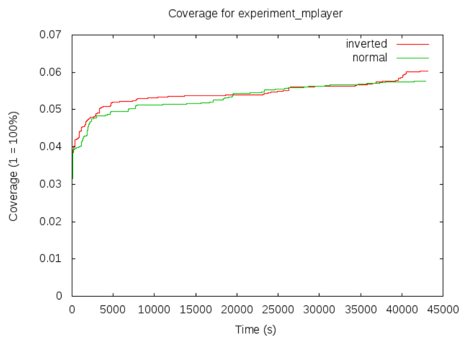


Figure 2.9: Coverage for `mplayer`

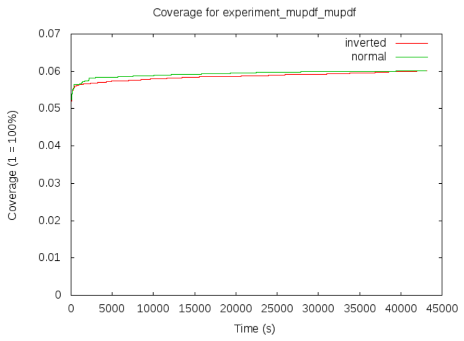


Figure 2.10: Coverage for `mupdf` from `mupdf` converting a PDF containing an image

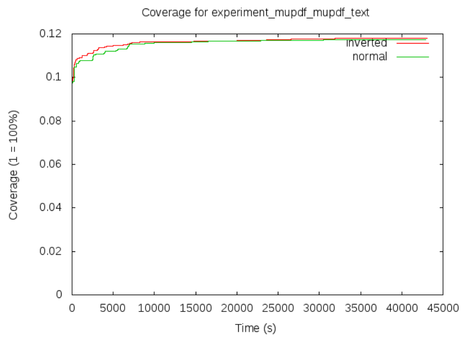


Figure 2.11: Coverage for `mupdf` from `mupdf` converting a PDF containing text

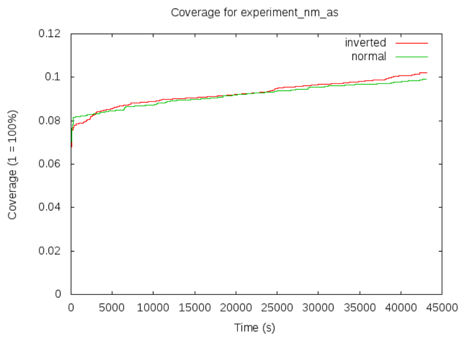


Figure 2.12: Coverage for `nm` with `as` as the inverse

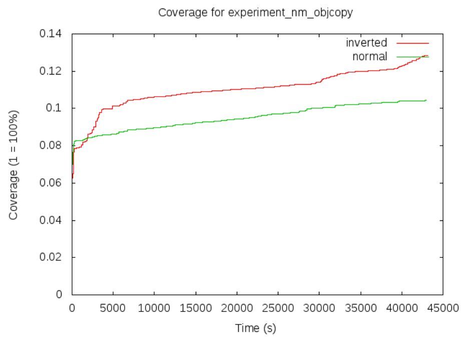


Figure 2.13: Coverage for `nm` with `objcopy` as the inverse

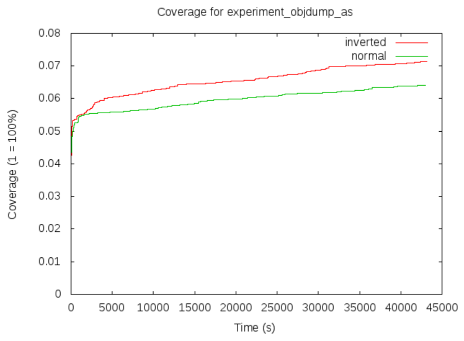


Figure 2.14: Coverage for `objdump` with `as` as the inverse

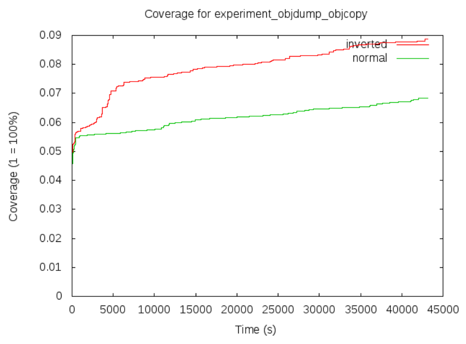


Figure 2.15: Coverage for `objdump` with `objcopy` as the inverse

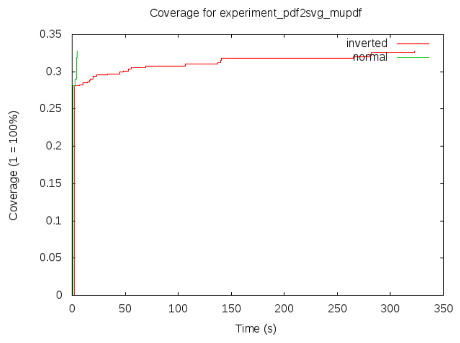


Figure 2.16: Coverage for `pdf2svg`

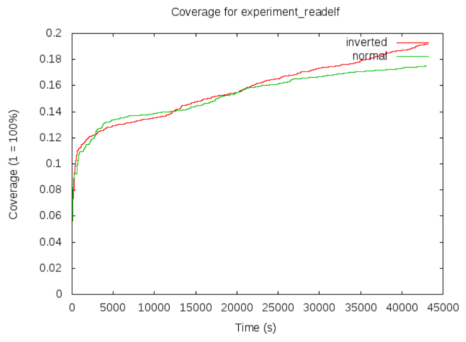


Figure 2.17: Coverage for `readelf` with `as` as the inverse

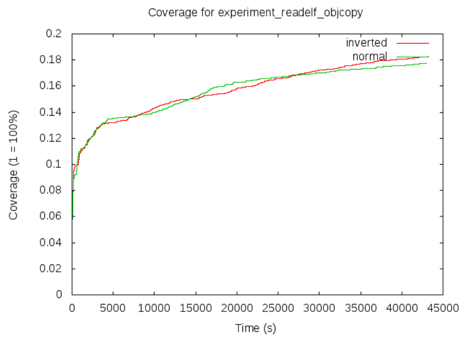


Figure 2.18: Coverage for `readelf` with `objcopy` as the inverse

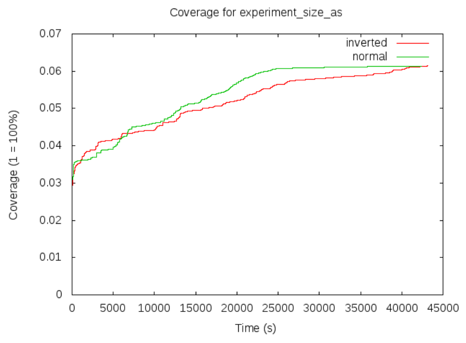


Figure 2.19: Coverage for `size` with `as` as the inverse

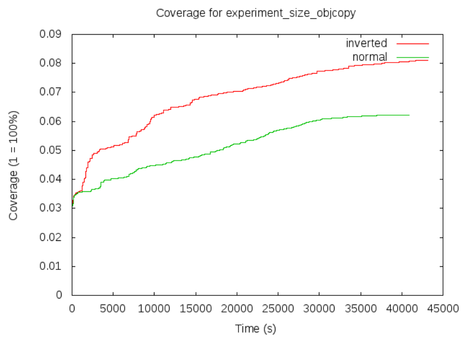


Figure 2.20: Coverage for `size` with `objcopy` as the inverse

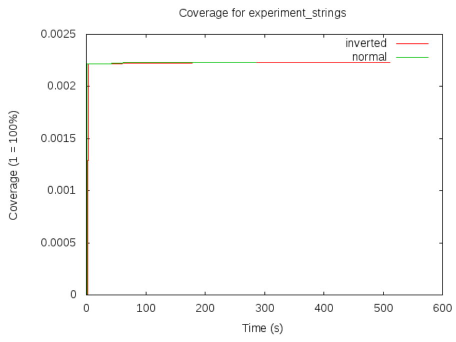


Figure 2.21: Coverage for `strings`

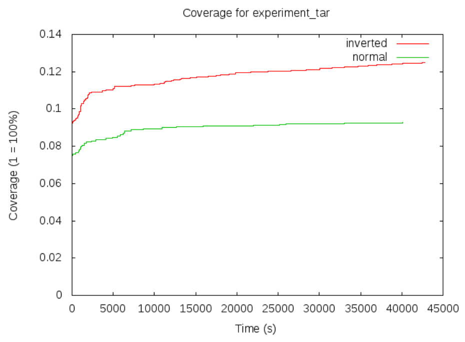


Figure 2.22: Coverage for `tar`

tion occurs for programs for which AFL reaches the maximum coverage (at least for the command line arguments it uses to run the test program) very quickly, so that any extra work that the inverse fuzzer does only slows it down. The tests of the Binutils programs also underscores the importance of using inverses whose functionality corresponds to that of the test program. While the assembler, `as`, does create binaries that Binutil programs can read, most Binutil programs only deal with data besides the machine code that the assembler creates, so fuzzing the assembly logic is generally not useful. Thus the only program where `as` was noticeably helpful was `objdump`, which performs disassembly. On the other hand, `objcopy` focuses on manipulating the other data that the Binutil programs parse, and therefore is more useful as an inverse program.

Bugs

We manually examined some of the bugs that the inverse fuzzer found and the normal fuzzer did not. Below are bugs we have found that have not been previously fixed.

a timestamp.

`c++filt`: The inverse fuzzer found a `NULL` reference in `c++filt` due to invalid error handling. Listing 2.1 shows the immediate cause of the fault. Even though the index is within the bounds of the array, `from->btypevec`, according to `from->numb`, an entry in the array is a `NULL` pointer. The offending entry is created in the function `register_Btype`, which increments the array size, and makes the new, last entry `NULL`. This would normally not be a problem, because the `remember_Btype` function puts a string in the last entry. But in case of an error, the caller of both functions returns after creating the new entry, but before populating it, as shown in Listing 2.3.

Listing 2.1: The `NULL` reference in `c++filt`

```
static void
work_stuff_copy_to_from (struct work_stuff *to,
                         struct work_stuff *from)
  ...
  for (i = 0; i < from->numb; i++)
    {
      int len = strlen (from->btypevec[i]) + 1;
      ...
    }
  ...
}
```

Listing 2.2: Creating the `NULL` entry in `c++filt`

```
static int
register_Btype (struct work_stuff *work)
{
  int ret;
  ...
  ret = work -> numb++;
  work -> btypevec[ret] = NULL;
  return(ret);
}
```

Listing 2.3: Incorrect error handling in `c++filt` keeps the `NULL` entry

```
static int
demangle_qualified (struct work_stuff *work, ...)
{
  ...
  int bindex = register_Btype (work);
  ...
  if (!success)
    return success;

  ...
  remember_Btype (work, temp.b, LEN_STRING (&temp), bindex);
  ...
}
```

`cpio`: While most of the apparent hangs that AFL found are false positives, the inverse fuzzer did find a real hang in `cpio`, where the normal fuzzer did not. Due to the fact that triggering the bug required a special mode, it was unlikely that the normal fuzzer would ever find it.

`cpio` hangs when it tries to create a file by opening it, as show in Listing 2.4. The problem is that when the file already exists, and is a named pipe, the function waits for another party to open the pipe on the other end [64]. In most modes such a case would only occur if the attacker was able to create the pipe after `cpio` checked for the existence of the file, and before it tries to open it. But for two non-default modes, `arf_newascii` and `arf_crcascii`, as shown in Listing 2.5, the attacker can use `cpio` itself to create the pipe after the check. The attacker can craft a file so that it defers the creation of the pipe and a regular file of the same name, so that `cpio` creates the pipe only after it checks for its existence, and then tries to create the regular file without checking. The normal fuzzer did not find this bug, and it is unlikely that it would, because the two modes require 6 magic bytes to match, as shown in Listing 2.6. And even if the file contained matching magic bytes, the rest of the file must still adhere to the specifications of the mode.

Listing 2.4: The code where `cpio` hangs. This function contains no check for the existence

of the file.

```
static void
create_final_defers ()
{
  struct deferment *d;
  ...
  for (d = deferments; d != NULL; d = d->next)
    {
      ...
      out_file_des = open (d->header.c_name,
                           O_CREAT | O_WRONLY | O_BINARY, 0600);
      ...
    }
  ...
}
```

Listing 2.5: Conditions for when cpio defers creating files, therefore bypassing existence checks.

```
static void
copyin_regular_file (...)
{
   ...
      if (file_hdr->c_nlink > 1
          && (archive_format == arf_newascii
              || archive_format == arf_crcascii) )
        {
          ...
              defer_copyin (file_hdr);
          ...
        }
   ...
}
```

Listing 2.6: Conditions for when cpio chooses a subformat that supports deferring file creation.

```
  if (!strncmp (tmpbuf, "070701", 6))
    archive_format = arf_newascii;
  ...
  else if (!strncmp (tmpbuf, "070702", 6))
    {
      archive_format = arf_crcascii;
      crc_i_flag = true;
    }
```

Finally, the inverse fuzzer also found a stack overflow in pdf2svg, but that was due to a bug in a library that is no longer in the most recent version.

Fuzzing ClamAV

For each test that used the inverse fuzzer, we took the seeds that the inverse generated, removed the upper quartile of the seeds in terms of file size, and used them as the seeds for fuzzing `clamdscan` with AFL. Like previous attempts to fuzz ClamAV, we did not fuzz the full version of `clamdscan`, but only the core functionality, to avoid the overhead due to setup [88].

Before fuzzing, we passed the seeds through `afl-cmin`, which approximately minimizes the number of seeds, while still covering all of the code that the original set covered [110]. After fuzzing for 5 days, we found no crashes, but the generated inputs gave us the data for training the monitor for ClamAV in the next chapter.

Conclusion

By automatically manipulating the programs used to generate seed inputs, we improved not only the code coverage of a state-of-the-art fuzzer, but also increased the number of bugs found, including some that would likely not have been found given only the manually-generated seed.

Chapter 3

Asymmetrically Monitoring an Intrusion Detection System on the Same Host

This chapter introduces a general intrusion detection system that can be automatically trained to protect another intrusion detection system. Unlike the symmetric monitors in the next chapter, this one follows the traditional architecture in which a monitor protects a resource, but not the other way around. This leaves the problem that there will be at least one monitor that is unprotected – creating a new monitor to protect the previous one will add a new, unprotected resource. That means that we need to focus on reducing the probability of bugs in the new monitor.

In spite of the weakness of the traditional architecture, sometimes it is the most feasible one. It is often the only option when both the monitor and the target processes are running on the same host. While Chinchani *et al.* do develop a single-host defense architecture in which multiple monitors protect each other in a cycle, it only works for the FreeBSD operating system, while other operating systems, such as Linux, do not have mechanisms for processes to monitor each other cyclically [17]. Nevertheless, the defense in this chapter, and previous single-host defenses have the advantage that they can use the resources that the host platform provides to detect most attacks more quickly, usually almost instantaneously [85, 17].

Like previous intrusion detection systems that monitor processes, the monitor in this chapter will develop a state machine of the system calls that the target process makes [104, 35]. But unlike prior models, the new state machine will avoid inferring the state of the program by trying to determine its location in the code, which, so far has relied on reconstructing the control flow graph of the program, or reconstructing the call stack of the process, which would rely on the call stack following a particular convention. The principle behind this chapter is that a more simple algorithm will be less likely to contain programming bugs (although the classification performance may suffer). More concretely, stack tracing algorithms have not been without bugs, such as a recent one in the Breakpad debugging framework, in which a specially crafted stack could give the illusion of being too deep for the stack tracer's buffer [76]. While it is easy to fix such errors, the fact that such a bug was only found recently in a tool that has been in development for over 10 years suggests that such pitfall is a persistent risk [1].

We can simplify the model by considering particular properties of an intrusion detection system. They are continuously-running processes that consist of a setup phase and infinite repetition of a main loop. The setup phase does not accept input, so we can unconditionally allow it to perform privileged operations. The main loop does accept input, which means that we need to be more restrictive there. While some parts of the main loop may still require privilege, not all of them do. In theory, the main loop of an IDS contains a few distinct steps: data collection, classification, and response [23]. While data collection and response are the most privileged steps, most of the bugs are in the classification step, more precisely when a parser tries to convert the raw information from the data collection into features that the core logic of the intrusion detection system can classify as malicious or benign. Of course, the real steps in the main loop of a real IDS may not correspond exactly with the three aforementioned ones. But it is nevertheless useful to discover the common steps, and separate the privileged and the unprivileged ones. Thus we have two goals: to differentiate between setup and looping, and to separate the privileged and unprivileged steps inside the loop iterations. The question then is: where are the boundaries between the steps? Assuming that there are common steps that exist for every iteration of the intrusion detection system's main loop, identifying the boundaries would involve identifying the loop iterations, and finding common system calls between them. The solution that this chapter introduces is to let the training data know when the intrusion detection system is receiving input.

The rest of the chapter will describe the threat model, how to train and use the system call model, implementation details, and experimental evaluations of the performance and accuracy of the models.

Threat Model

The attacker is assumed to be an outsider who is trying to gain access through a running IDS. In particular, the attacker can influence the IDS by controlling its input. But the attacker has not previously gained access to the IDS's host through other means, and therefore cannot control or disable the IDS directly.

System Call Model

The models for the system calls will resemble finite state automata, where the inputs are the system calls. But the goal of the model is to find and represent the stages of an intrusion detection system using unambiguous boundaries. Thus, each state is only associated with a single system call, which signifies the entry into the state, and contains a set of allowed system calls. Some of the allowed calls are transitions into their respective states, and the rest of which do not change the state. To prevent the model from being too coarse grained, in each state, the repetition of certain system calls can cause the monitor to issue an alert, even if the model classifies a single instance of that call in that state as normal. We now describe the parts of the model in more detail.

System calls

A system call consists of a number that identifies the functionality, and a limited list of parameters, depending on the identifying number [65]. In general, variations in the parameters are expected, so it would require extensive training to discover a pattern amongst all the different combinations of paramaters. But there are a few types of parameters where a process will only use a few, discrete values. For those, it is possible to include their exact values when modeling a system call. These types are:

- Enum: Certain system calls support a limited number of options. For example, `lseek` is a system call for navigating to positions inside a file, and the last parameter determines whether to calculate the new position relative to the beginning of the file, the current position, or the end of the file [61]. Other times, the value is a command to a particular resource, which is the case for `fcntl` [60].

- Constant strings: In general, there is no limit to the contents a string can store, and even if there was only a small number of strings, determining its value requires reading and copying memory buffers whose lengths are not known ahead of time. So the model will consider only constant strings whose values are stored in the program binary [59]. This not only limits the number of strings, but also requires knowing only its location inside the program, instead of its value.
- Flags: The model treats parameters as a flags type if it contains bits whose meanings depend on whether they are set or cleared. In principle, a flag can have as many values as an integer with the same number of bits. In practice, as the experiments will reveal, a program will only use a limited number of them.

Currently, the parameter types are manually created.

States

A state consists of a number of system calls of the following types:

- The entry call: The system call whose appearance indicates entry into this state. Each state can have at most one entry call. Effectively, if we ignore states without entry calls, then a process has a one-to-one correspondence between entry system calls and states. Besides shrinking the size of the model, by limiting the number of states per system call to at most one, such one-to-one correspondence makes the model more resilient in the face of false positives: The monitor can reorient itself back to a state as soon as it observes an entry system call, even if the system call is not allowed in the current state.

- Internal calls: Internal system calls are system calls that the process can make, while in a particular state, and still remain in the same state. Internal calls reflect the notion that each step in the process's execution could allow several different system calls, without entering the next step. However, even if a step allows a system call, it might not allow the process to make the same sensitive system call multiple times. Thus each state indicates which internal calls the process can repeatedly call, before making an entry call. If the model was a true finite state automaton, each unrepeatable system call would double the number of states associated with a step: in half of them, the process has not made the system call, and is therefore still allowed to do so; in the other half, it has already made the call, and cannot do so again.
- Transitions: a transition is an entry system call that a process can make in a state. A state can contain a transition to itself.

Thread Rule

A thread rule is a set of states for a thread or set of threads. It contains exactly one state that does not have an entry call. In other words, if we use a dummy state for this special state, there is a one-to-one correspondence between all states and entry calls in the thread rule.

A program can have multiple thread rules, and the last one may be general, which means that it can apply to multiple threads. This is useful for when a program spawns multiple threads running the same code. How we determine the need for a general thread will be explained when we describe the training algorithm in the next subsection.

Building a Model

Training a model consists of a recording and a training step. The first step involves running the intrusion detection system over multiple test inputs and recording system calls. The second step takes the system calls to find entry calls and states. The training is tailored for the property that real-time intrusion detection systems act like servers in the sense that they ideally run forever, and execute a main loop that iterates over inputs as they arrive. Since the defense is supposed to work against attackers who exploit the input procesing part of the IDS, it is more useful to focus on those steps. In addition, as noted in Chapter 2, some intrusion detection systems have a long setup time. So by isolating the system calls the process makes while processing input, we can focus on the part of the execution that is both sensitive and relatively inexpensive to process. One additional challenge in modeling programs that process outside data on demand is that they often create different threads. If each thread ran different code, we can simply train them separately. That is because we can be sure that the number of threads is bounded, since the code size is bounded. But some threads could spawn during different times to process some of the inputs, before terminating. So it would be more accurate and scalable to create a common model for threads of the latter type.

Both the recording and training steps address the problem of estimating the input boundaries, while the second step also decides which threads to group together.

The recording process runs as follows:

1. Start running the IDS, while tracking its system calls.
2. Repeatedly try to send an input to the IDS, until it indicates that it has processed input, and therefore is running. Denote these inputs as probing inputs. In our examples, the input will be one that the IDS will consider malicious, so this step will check if the IDS has detected it.
3. Send the inputs from a corpus to the IDS, and in the system call log of each thread that is still running, record when the input was sent. Wait for a short interval (10 seconds in the experiments) between each input, in order to give the IDS the chance to process it.
4. Wait for a long interval (30 minutes in the experiments), to record system calls during periods of inactivity.
5. Send the inputs from the corpus again, but as fast as possible.

In order to infer the need for a general thread, the training step takes advantage of the following heuristic: If a unique thread only ran for a limited time, the IDS would be inconsistent for the same input at different times. Threads that run similar code, but for only specific inputs only appear when new input requires them. That means, not only should a unique thread run indefinitely, but it should also run ahead of all the threads that run the same code, but for different inputs. The processing therefore starts by categorizing the inputs according to Split, as shown in Algorithm 5.

The entry points we choose should be common to all inputs, but not appear in the setup stage. So they should be an intersection of the sets of system calls that are sepa-

Algorithm 5 The algorithm for grouping threads' system call logs

```
procedure Split(logs) ▷ logs is a list of per-thread log files, ordered by appearance time.
    maxin ← $-\infty$                          ▷ maxin will hold the index of the last input
    for log $\in$ logs do
        in ← The index of the last input in log
        if in $>$ maxin then
            maxin ← in
        end if
    end for
    groups ← $\emptyset$                              ▷ groups is the list of thread groups.
    general ← $\emptyset$                           ▷ general is the list of general threads.
    for log $\in$ logs do
        if The first non-probing input or the input indexed at maxin is not in log or general $\neq \emptyset$ then
            general ← general $\cup$ {log}
        else
            groups ← groups $\cup$ {{log}}
        end if
    end for
    if general $\neq \emptyset$ then
        groups ← groups $\cup$ {general}
    end if
    return general $\neq \emptyset$, groups    ▷ Return the thread groups, as well as whether or not the last one is for general threads.
end procedure
```

rated by when the inputs occur, minus the system calls that occur before the last probing input, which is the one to which the IDS first responded. The set subtraction step can be performed per thread, since a system call subtracted from one set will not reappear in the intersection of the reduced set with another set. But the heuristic does not work if the IDS does not react to the inputs quickly enough: it would possible that there will not appear to be any possible entry calls at all. One possible cause is that when the inputs are fed in as quickly as possible, regardless of whether the IDS has processed the last input or not, there will be some inputs for which there are apparently no system calls at all, because the IDS has not performed any processing. So when the intersection is empty, retry, but only calculate the intersections of the system calls induced by feeding the inputs with a pause between each one. The second possibility is that the IDS could have detected some of the earlier probing inputs, but not quickly enough for the probing procedure to have detected it, so it would appear that some of the system calls that should be used only for processing inputs appear to be used in the setup. So when a log yields no entry calls, retry, but without subtracting any calls.

With a set of entry calls and logs of ordered system calls for the appropriate group of threads, the training phase can finish by generating the respective rules. The main data structure will be a mapping, which keeps track of whether a system call has appeared zero, once, or many times during a state. Algorithm 6 describes the rule generation algorithm, GenRules.

Enforcing the Model

The monitor tracks the system calls that the monitored process makes, and compares them against the model. When it finds an entry call, its state switches to the corresponding state. It will also check if the system call is allowed by the state. If not, it will issue an alert.

More specifically, for one thread, given the rules, T:

1. Initialize the state, s, to $\perp$.
2. Initialize the set of system calls that cannot be repeated, R, to $\emptyset$.
3. For each system call, c
    a) Let S be the rules for c in T.

Algorithm 6 The algorithm for generating a thread rule

```
procedure GenRules(logs, $E$)    ▷ logs is the list of per-thread logs in a group; $E$ is the set of entry calls
    $T \leftarrow \emptyset$    ▷ $T$ maps each entry call to a state's rules
    for $e \in E$ do
        Let $T$ map $e \leftarrow \emptyset$
    end for
    for log $\in$ logs do
        $e \leftarrow \perp$    ▷ $e$ is the current entry call. It is initialized to be a dummy value representing the starting state
        $S \leftarrow$ the rules for $\perp$ in $T$    ▷ $S$ maps a system call to whether it has repeated or not. If a call is not in the domain, that means that it has not appeared at all.
        for $c \in$ log do
            if $c \in E$ then    ▷ Found a transition into another state
                Let $S$ map $c \leftarrow$ false
                Let $T$ map $e \leftarrow S$    ▷ Save the previous state
                $e \leftarrow c$
                $S \leftarrow$ the rules for $c$ in $T$    ▷ Fetch what we already know about the next state. It might be empty
            else    ▷ Keep the same state, and only check for a repeat.
                if $c \in S$ then
                    Let $S$ map $c \leftarrow$ true
                else
                    Let $S$ map $c \leftarrow$ false
                end if
            end if
        end for
        Let $T$ map $e \leftarrow S$    ▷ Save the last state in the log
    end for
    Return $T$
end procedure
```

        b) If c is a key in S:

            i. S maps c to `false`

                A. If c is in R, issue an alert. Otherwise add c to R.

            Else, issue an alert.

        c) If c is an entry call in T, set s to c, and reset R to $\emptyset$.

Now we're left with the top level decision of which thread rules to use for a new thread. Suppose there are N thread rules, T_1 to T_N.

1. Initialize the thread counter, t, to 1.
2. Whenever the task spawns a new thread:
    - a) If $t > T$:
        - i. If the last thread rule was created from multiple logs, output T_N. Otherwise, issue an alert.

      Otherwise, output T_t
    - b) Increment t

Implementation

The implementation of both the system call recording and monitoring programs are based on the `strace` system call tracing program [115], which in turn uses the `ptrace` system call, which is used for monitoring programs while they are running [63]. The training phase is written in Python3. Using `sloccount`, the C code for gathering and monitoring system calls took 2697 lines of code, while the training script had 846 lines.

Evaluation

We test two existing intrusion detection systems that are used to protect different settings. We measure the performance impact of the monitor, as well as number of false positives, and quantify how restrictive the trained models are.

Test Programs

We tested a host-based IDS, ClamAV, and a network-based IDS, Snort [1]. In particular, for ClamAV, we monitored the `clamd` daemon, and sent it data using `clamdscan`. Its version is the same as the one we fuzzed. For Snort, we only used the `snort` program, version 2.9.7.0, while using a packet replay script to feed it data. Recall that we do not need the source code of the IDS. In fact, for Snort, we used the binary that we could install on the Ubuntu.

For ClamAV, we used the same virtual machines from the previous chapter. For Snort, we isolated it in a virtual machine running on a laptop computer. The virtual machine has a single, 2.40 GHz processor, while the laptop has 4. The virtual machine is not directly exposed to the Internet, but is instead inside a local area network that only consists of the host system and utilities of VMWare.

Data Sources

For ClamAV, the inputs we used for training and testing the monitor are taken from the inputs that the fuzzer generated. For Snort, we recorded 1000 packets of local network

[1]https://www.snort.org/

Table 3.1: Training time on full data

Program	Data Collection time (s)	Training time (s)	Total time (s)
ClamAV	18000	18.163	18018.163
Snort	12624	1.162	12625.162

traffic. During the experiments, the IP address of the recording host would be replaced by the IP address of the current host, and all other IP addresses were replaced by a different IP address that is on the same subnet. For detecting if ClamAV is running, we used the EICAR test file as the probing input. This is a file that anti-virus programs should recognize as a virus [54]. When `clamdscan` sends the known virus to an online `clamd`, the former program will terminate with an exit code 1. For detecting if Snort is running, the replay script plays a ping packet. If snort is running, it will create an entry in the alerts log.

Performance Cost

To measure the performance cost, we trained the models for both programs using the full available data. The total sampling and training times are shown in Table 3.1.

For ClamAV, we had `clamdscan` send the generated files to `clamd` one after the other. Since `clamdscan` only terminates after receiving a result for a file, the rate that `clamdscan` executes therefore is proportional to the speed of `clamd`. It turns out that the performance of `clamd` improves in the early part of its runtime, which means that the performance will be worse for the initial batch of files we send it. Thus we sent the files twice. We ran 10 trials for each setting. The results are shown in Table 3.2. The speed slows down by a factor between 2 and 3 in the first round, and almost 2 in the second round. But when we compared it to GDB and `strace`, where they trace the unused `pwritev` system call, we

Table 3.2: Speed of ClamAV for different monitoring settings

Round	Unmonitored	Monitored	GDB	`strace`
First Round	260.09	101.16	74.22	251.74
Second Round	507.79	297.92	232.84	444.98

Table 3.3: Speed of Snort for different monitoring settings

Metric	Unmonitored	Monitored
Packets	308507	307491
Alerts	462575	461243

see that the performance is not unusual for system call tracing tools.

For Snort, we directly count how much data `snort` itself has processed. That is because, unlike `clamdscan`, our replay script does not wait for Snort to finish processing each packet. Thus we pinged the virtual machine 100 times per second for 30 minutes, and counted how many packets and alerts Snort processed with and without the monitor. The results are shown in Table 3.3.

The overhead for Snort is less than that of ClamAV. The reason might be the fact that the model for Snort is simpler. The file containing the rules is only 2608 bytes, while the file for the ClamAV rules are 4412. The difference in model complexity also suggests that ClamAV makes more system calls per input. In the next subsection, we will count the total number of system calls, which will confirm this guess.

Precision

To measure precision we split the data between training and testing sets. Concretely, we train the model on a random tenth of the data, and test it on the remaining data.

Since none of the inputs are attacks against the intrusion detection systems themselves, any alerts are false positives. We count alerts two ways: the number of system

Table 3.4: Training time on one tenth of data

Program	Data Collection time (s)	Training time (s)	Total time (s)
ClamAV	3425.18	4.73	3429.91
Snort	2878.07	0.32	2878.38

Table 3.5: False positive rate by system calls

Program	Number of alerts	Number of system calls	False Positive Rate
ClamAV	1081	1996891	0.000541
Snort	0	41160	0

Table 3.6: False positive rate by inputs

Program	Number of alerts	Number of inputs	False Positive Rate
ClamAV	108	14510	0.007443
Snort	0	9010	0

calls, and the number of discrete inputs (files or packets) that are reported. To classify a system call as an alert, we classify the monitor to be in an alert state if it has encountered an invalid system call, and has not yet encountered an entry call, which would let it transition to a valid state. Any system call that the monitor detects during the alert state, including the first invalid call, is counted as reported. Similarly, if the monitor is ever in an alert state between the time an input is sent to the IDS, and the time that the next input is sent, then the input is counted as reported.

We ran 10 trials for each program. Table 3.4 shows the average build time with the smaller training sets. Table 3.5 shows the false positive rates counted by system calls, and Table 3.6 shows the false positive rates counted by inputs,

In both cases, ClamAV has a false positive rate of less than 1%, while Snort has no false positives at all.

The number of system calls and inputs in the tables also show that ClamAV indeed uses many more system calls per input than Snort. ClamAV makes an average of 137.6

system calls per input, while Snort only makes 4.6.

Completeness

A low false positive rate is not useful if a model is too permissive. As previously mentioned, we should be able to isolate between the setup step and main loop, and discern privileged and unprivileged states in the main loop. To quantify how permissive a model is, we look at the models trained on the full data, and count the number of unique system calls that each thread rule allows, and the maximum number of unique system calls a non-starting state allows. The total number of system calls shows how permissive each thread rule is, in general. The maximum number of system calls per state shows how permissive the thread rule is at any point in time. We exclude the starting state, because the monitor will never enter that state due to actions by the adversary, who can only control the inputs. We also count the number of sensitive states in each thread rule. A sensitive state is a non-starting state that allows system calls that access and manipulate files, or execute a new program. We do not count spawning a new thread or process as necessarily sensitive, because the monitor is still able to track the child.

ClamAV has three thread rules, and the last one is a general rule set. The total numbers of permitted system calls are 56, 9 and 40. The maximum numbers of permitted system calls in a non-starting state are 10, 1 and 27. The initial states allow 53, 7 and 6 system calls. The thread rules have 2, 5 and 8 non-starting states, respectively, and out of those, 1, none and 5 states are sensitive. Unfortunately, the last thread rule is not restrictive. That is because the threads that follow it are only spawned when `clamd` receives input.

So most of the system calls will be used for parsing inputs, and there is no real setup step that does most of the privileged tasks.

Snort has two thread rules, and none of them are general. The total numbers of permitted system calls are 83 and 4, while the maximum numbers of permitted system calls in a non-starting state are 4 and 1, and the starting states allow 82 and 4. In fact, as speculated previously, the model is indeed small, which makes it restrictive. It only has two states in each thread rule, and the system calls that `snort` makes after the initial states are `read`, `fstat`, `restart_syscall`, `poll` and `nanosleep` [65]. In other words, Snort does not access any new resources after setup.

So our model cannot prevent the IDS from making sensitive system calls, but we can isolate them in a limited number of states.

Conclusion

We have developed a new technique for modeling system calls for detecting anomalies in program behavior. The technique is based on separating the operation of intrusion detection systems into stages: the setup and processing stages, and substages during the processing of each input.

The models achieve small false positive rates, while reducing the number of system calls the attacker can make at any time.

Chapter 4

Symmetrically Monitoring Network-Based Intrusion Detection Systems

This chapter addresses a weak point in the previous chapter, namely that when the monitor runs on the same host as the monitored resource, an adversary with physical access could simply shut down the host. The solution will use multiple monitors that protect each other in a cyclic fashion. The network requires an adversary to not only perform multiple attacks, but also do so in a small amount of time. This chapter first develops a theoretical model to evaluate a network of mutually-monitoring monitors. Next we describe a protocol that applies its principles. Finally, we evaluate the model by simulation in a large setting, and experiment on the protocol in a small setting.

Threat Model

In this chapter, the threat model is defined relative to available monitors that we will be combining. As long as the available monitor is monitoring the attack's target, it will detect the attack within a given time. We also assume that messages between monitors are reliable, secret, and unforgeable, which we can enforce with existing means, such as using a TLS connection.

In our concrete example, the attack model is a complement to the attack model of the previous chapter. Instead of trying to subvert the original IDS, it can kill its process, or shut down the host. The monitors we will be using are failure detectors, which will detect if the original IDS process has terminated in near-instantaneous time, or if another host is no longer running, within a configurable interval.

Modeling Adversary Cost

In the worst case, we can consider the adversary as a single entity that can coordinate multiple attacks at the same time. To enable simultaneous attacks, we consider the entities that perform them not as independent attackers, but as processors that carry out the tasks assigned by the adversary, which, in this case, are the assets to attack.

We deliberately relate the abilities of the adversary to processors, and the victim assets to tasks, because there exist numerous problems in graph and complexity theory concerning the processing and scheduling of tasks, under restrictions due to interdependencies and time constraints. The adversary can consider planning an attack against a defensive graph as such a scheduling problem, with an attack against an individual asset corresponding to a task. However, in our model, dependencies are not as strict as in traditional dependency graphs: The adversary does not necessarily have to complete its attacks against all monitors defending a target before attacking the target itself; it only has to finish its attacks against the defenses before they detect the attack against the target.

We will thus model the defense graph as what we call a *delayed* dependency graph, with the assets as the nodes, and the monitoring relationships as edges, so that nodes with outgoing edges represent monitors (such as failure detectors), and nodes without them are the basic assets we ultimately want to protect (such as user applications and virtual machines). We will represent the adversary's power as its number of processors, each of which can perform one attack at a time. Finding the exact number of processors that the adversary will always need is therefore a generalization of the multiprocessor scheduling problem, which is why it is NP-hard. On the other hand, the average number of processors is not only a tractable lower bound, but is also adaptable for an adversary whose power is less known and more elastic.

Graph Model

In a delayed dependency graph, $DDG = (V, E, l, d)$, V is the set of nodes or tasks, $E \subseteq V \times V$ is the set of directed edges, $l : V \to \{x \in \mathbb{R} : x > 0\}$ denotes the length of time to complete each task, which is inversely related to the speed of the adversary, and $d : E \to \{x \in \mathbb{R} : x \geq 0\}$ represents the dependency delay, which is inversely related to the speed of the defender. In the context of monitors, if $(u, v) \in E$, that means that u is monitoring v, and it takes $d(u, v)$ units of time for u to detect an attack on v.

A delayed dependency schedule of m processors, $(\sigma, \mathcal{A})$, consists of the start time assignment, $\sigma : V \to \{x \in \mathbb{R} : x \geq 0\}$, and a partition of a subset of the nodes into processors, $\mathcal{A} = (A_1, A_2, \ldots, A_m)$, *i.e.* $\forall p \in [m], q \in [m]$ so that $q \neq p, A_p \cap A_q = \emptyset$. For simplicity of notation, we can define the absolute completion time of each node, $c_\sigma : V \to \{x \in \mathbb{R} : x > 0\}$, as $c_\sigma(v) = \sigma(v) + l(v)$. If a task, v, is never performed, *i.e.* $\forall p \in [m] : v \notin A_p$, then we say that $\sigma(v) = c_\sigma(v) = \infty$.

The following properties must hold:

1. Schedule starts at time 0: $\exists v \in V$ so that $\sigma(v) = 0$
2. At most one node per processor at a time:

$$\forall p \in [m], u \in A_p, v \in A_p \text{ so that } u \neq v :$$
$$c_\sigma(u) \leq \sigma(v) \text{ or } \sigma(u) \geq c_\sigma(v)$$

3. Delayed dependency: $\forall (u, v) \in E$:

$$c_\sigma(u) \quad \leq \quad \sigma(v) + d(u, v)$$

We will find it useful to define the (possibly negative) start delay of an edge, $s : E \to \mathbb{R}$, so that $\forall (u, v) \in E : s(u, v) = d(u, v) - l(u)$. This function denotes the latest time that u can start after v. We can express the delayed dependency requirement equivalently to the above inequality:

$$\begin{aligned} c_\sigma(u) &\leq \sigma(v) + d(u, v) \\ \sigma(u) + l(u) &\leq \sigma(v) + d(u, v) \\ \sigma(u) &\leq \sigma(v) + d(u, v) - l(u) \\ \sigma(u) &\leq \sigma(v) + s(u, v) \end{aligned}$$

According to our model, as shown in Figure 4.1, the tail node, from which the bold edge points, represents the monitoring node, and the head node, to which the bold edge points,

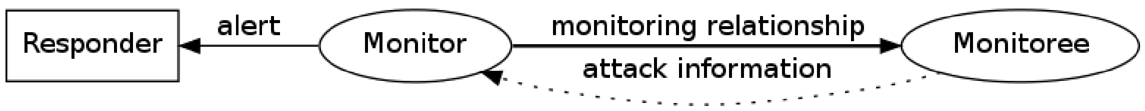


Figure 4.1: The bold edge is the monitoring relationship, as represented in the graph. The dotted edge is the implicit flow of attack information. To illustrate the delayed dependency, the thin, solid edge represents the explicit alert issued to an entity responsible for the response, if the adversary is unable to attack the monitor in time. The last two edges are for illustrative purposes only; they are implied by the first edge, and therefore not explicitly represented in the model.

represents the protected asset. There were three conflicting considerations in choosing the direction of the edge, and we chose the direction that agreed with the majority. Intuitively, an outgoing edge suggests that the monitor at the node is actively engaged in the monitoring relationship that formed the edge. On the other hand, the monitored node does not necessarily have to actively maintain the relationship. Moreover, in a traditional dependency graph, the tail node represents the task that must be completed before the task at the head node can be started. While delayed dependency does not necessarily require this order, it does require that the task of the tail node is completed no later than a specific delay after the task of the head node has been started. This choice of direction, however, conflicts with the notion of the passive propagation of the "obligation" to complete an attack, as well as the implicit transmission of information about an attack to the monitoring node. That is, with our choice of direction, attack information flows upstream, and thus the adversary must attack upstream nodes to stop this flow. However, not only is this intuition in the minority, but it is also an implicit notion that may not practically appear: The monitored node might not actively send information to its monitor, and we want to design a graph so that the adversary will be unable to fulfill its obligations.

This model does not explicitly account for tasks that can be subdivided and performed in non-contiguous time intervals. However, if each task has up to a polynomial number of subtasks, so that pausing the attack at the end of each subtask preserves progress, a task, v, consisting of k subtasks, $\{v_1, \ldots, v_k\}$, can be subdivided so that v_1 keeps all the incoming edges of v, v_k keeps all the outgoing edges, and for every pair of consecutive steps, v_{i-1}, v_i, we enforce that v_{i-1} must be completed by the start of v_i by setting $d(v_{i-1}, v_i) = 0$. Fortunately, in the approximation, when we aggregate the task lengths, regardless of order or subdivisions, this trick will not be necessary.

Difficulty of Exact Metric

An adversary with m processors will have to choose an attack schedule in which its processors are enough to attack any nodes before they issue an alert. In other words, it must choose a schedule to meet all of the deadlines. We call this problem *Delayed Dependency*, and define it as follows: Given a delayed dependency graph, does there exist a delayed dependency schedule with m processors?

However, as we will prove in the appendix, this problem is NP-hard, even when we restrict the graph to constant degrees. For practical purposes, we will therefore need an approximation for which we have an inexpensive algorithm.

Average Metric

Given the complexity of the exact metric, we turn to an approximation: The average number of *active* processors the adversary will need during the attack campaign. Roughly speaking, we calculate it as the ratio of the total active time of all the processors to the maximum attack time.

The total active processing time depends on the nodes that the adversary must attack, including the original victim, in order to avoid detection. Suppose $v \in V$ is the first victim. Let $R_v \subseteq V$ be all the nodes that can reach v, including v itself. That means that the adversary must attack all $u \in R_v$ to avoid detection. Moreover, the adversary cannot simply attack all $u \in R_v$ at its leisure. For example, all adjacent nodes, a, where $(a, v) \in E$, must be attacked by time $s(a, v)$. Indeed, by induction, for all $u \in R_v$, the latest attack *start* time for u is the length of the shortest path from u to v, if we used s as the weight of the edges. And if we add $l(u)$ to the distance, since the first term in the distance was $s(u, \cdot)$, we can change it to $d(u, \cdot)$, so that we derive the latest time the attack on u must be *completed*. Let us denote this latter value by $t(u, v)$. Since the attack schedule does not end until the last node has been successfully attacked, the attack time is the maximum value of $t(u, v)$. An apparent exception arises if there is a $u \in R_v$ for which the distance is negative. But then that u must be attacked before v, so that v cannot be considered the first victim.

We can accordingly define a metric based on a particular starting node, and extend it to an adversary who wants to minimize that value:

Definition 1 *The average processor requirement for $v \in V$ is:*

$$M_v = \frac{\sum_{u \in R_v} l(u)}{\max_{u \in R_v} t(u, v)}$$

if $\forall u \in R_v, t(u, v) - l(u) \geq 0$*. Otherwise,* $M_v = \infty$

In the worst case (from the defender's point of view), the adversary is willing to take down any node first, so we accordingly derive the security metric for the entire graph:

Definition 2 *The average processor requirement for a delayed dependency graph is:*

$$M = \min_{v \in V} M_v$$

The algorithm for finding M is simply as follows:

1. Calculate the distances between the nodes using the Floyd-Warshall algorithm [18].
2. If the Floyd-Warshall algorithm detects any negative-weight cycles, remove all the nodes that are in or reachable from the cycles.
3. For each node $v \in V$:
   a) Find R_v, *i.e.* the set of all nodes with a finite distance to v.
   b) Calculate the workload $W_v = \sum_{u \in R_v} l(u)$.
   c) For every reachable node $u \in R_v$, calculate $t(u, v)$ by looking up the distance from u to v, and adding $l(u)$. If there exists $u \in R_v$, so that the distance is negative, set $t_v = 0$. Otherwise, let t_v be the maximum $t(u, v)$.
   d) Calculate $M_v = \frac{W_v}{t_v}$.
4. Output the minimum M_v.

The only part of the algorithm not mentioned in our definitions is the removal of negative-weight cycles. If the adversary ever attacked, or had to attack a node in a negative-weight cycle, then the adversary would never succeed, since all nodes in the cycle will force their starting times to be before whenever their starting times actually are. Note that for any path that includes a removed node, the last node would also be removed, because the last node is reachable from the negative cycle.

The most obvious benefit of this metric is its simplicity. Not only does there exist a polynomial-time algorithm, but by calculating work in aggregate, we can ignore any subdivision of work, thus removing the limit on subdivision we had before.

This average therefore lets us quickly approximate the exact processor requirement, but its usefulness lies in its ability to estimate real-life, uncertain environments, rather than the theoretical, exact value. In the exact definition, the adversary's number of processors must suffice for all times during the attack, so that the average number of processors is a lower bound. In fact this bound is tight. For a directed cycle of n nodes, with start delay and task length of 1, the farthest distance from every node is $n - 1$, so that the attack time is n. Since the graph is strongly connected, W_v is always n, so that the average processor load is 1. When we look at the delayed dependency problem with only 1 processor, we find that it is enough to take down one node after the other, along the edges of the cycle.

However, in general, this lower bound is not even asymptotically equal to the exact value. As a counterexample, consider a symmetric graph (*i.e.* we treat the edges as bidirectional) with $2h$ nodes, as illustrated in Figure 4.2. All tasks have lengths 1, and all edges have start delay of 1. $h + 2$ nodes are in a complete subgraph, and one of these nodes, as well as the $h - 2$ remaining nodes, form a bidirectional path. This graph is strongly connected, so that $\forall v \in V$, $W_v = 2h$. Thus we only need to find the maximum $t(u, v)$, which is the distance between any node that is only in the complete subgraph, and the node at the far end of the path, added to 1 for the completion time of either end. The

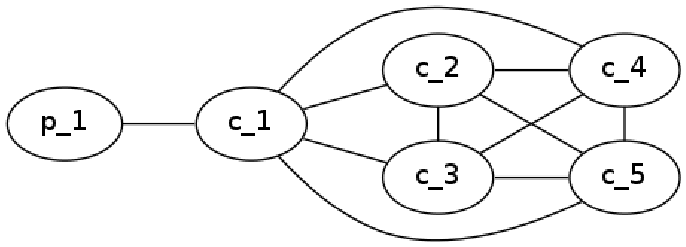


Figure 4.2: Counterexample to asymptotic equivalence between lower bound and exact value, with $h = 3$.

maximum $t(u, v)$ is therefore h, so the lower bound is a constant value, $\frac{2h}{h} = 2$. However, whenever a processor starts any node in the complete subgraph, the entire subgraph must be completed in 2 time units, so that during that time, an average of $\frac{h+2}{2}$ processors are necessary, so at least $\frac{h+2}{2}$ processors are needed for the attack.

Nevertheless, the average metric is preferable not only because there exists a polynomial-time algorithm to find it, but also because we can factor out some variables that we cannot necessarily deduce from the graph topology: The attack speed of the adversary's processors, and the detection time of the defender nodes.

Assume that every processor has the same attack speed, r_A, and every monitoring node has the same detection speed, r_D. We can think of $L(u) = l(u) \times r_A$ as the processor-independent amount of work required to attack a node. As for $t(u, v)$, it is no greater than the distance from u to v, if we used d as the weight, since d does not have the task lengths subtracted, as is the case with s. We denote the latter distance as $t_{\text{inst}}(u, v)$, which is the hypothetical value of $t(u, v)$ in the worst-case scenario of instantaneous attacks. Then $T_{\text{inst}}(u, v) = t_{\text{inst}}(u, v) \times r_D \geq t(u, v) \times r_D$ is a defender-independent upper bound on the work required to propagate attack information, which only depends on the diameter of the graph.

Then we can rewrite M to separate the values dependent on the graph, and the variables dependent on the defender and adversary:

$$\begin{aligned} M &\geq \min_{v \in V} \frac{\sum_{u \in R_v} \frac{L(u)}{r_A}}{\max_{u \in R_v} \frac{T_{\text{inst}}(u,v)}{r_D}} \\ &= \frac{r_D}{r_A} \min_{v \in V} \frac{\sum_{u \in R_v} L(u)}{\max_{u \in R_v} T_{\text{inst}}(u,v)} \end{aligned}$$

This form not only confirms the intuition that the defender should be fast relative to the adversary, but also that the graph itself should have a small diameter, and have as many nodes as possible reachable from each other. The last two ideas remain apparent when we multiply the processor requirement by the rate of each processor, which gives us the more intuitive and adversary-independent notion of power as processor-independent work over real time:

$$M r_A \geq r_D \min_{v \in V} \frac{\sum_{u \in R_v} L(u)}{\max_{u \in R_v} T_{\text{inst}}(u,v)}$$

As an additional benefit, the average measure also more closely represents a growing class of adversaries with elastic resources. The exact measure indicates the number of fixed-speed processors the adversary must keep to suffice for times when the maximum number of processors is required, even if most of the processors will stay idle for most of the time. The average case considers adversaries who are able to, for example, rent or overclock processors when they are needed, and release them or return them to their default speed when they are not, but cannot afford a large number of heavily loaded processors in the long term.

Defense Graph Design Principles

As we hinted in the previous subsection, we can optimize the security of a defense graph, regardless of the relative strengths of the adversary and defender. To maximize the numerator of the average processor requirement, we want to maximize the number of nodes that can reach each node. The number of reaching nodes is maximum when the graph is strongly connected. It does not have to be a single cycle, as suggested in [17], although that is the cheapest, strongly-connected topology. Of course, the intended roles of each node may preclude certain edges and topologies. If the nodes are to be used for anything other than mutual defense, then there will exist nodes –the basic assets that we originally wanted to protect – that will be unable to defend any of the other nodes, and therefore cannot reach the rest of the graph. In these cases, to make a pessimisstic estimate, one can only consider the strongly connected components that can reach the whole graph.

But we also want to minimize the diameter of the graph. The diameter is minimum if the graph is complete. However, this might be difficult to achieve in practice. In fact, it may not be worth it: If we can scale up the speed of the defender with proportional cost, the complete graph is no better than a cycle. For n nodes, the complete graph has $n(n-1)$ directed edges, and a maximum $T_{\text{inst}}(u,v)$ of 1, while the cycle has n directed edges, and a maximum $T_{\text{inst}}(u,v)$ of $(n-1)$. Thus, to achieve the same average processor requirement, the detectors in the cycle can be sped up by a factor of $(n-1)$, so that the new cost of the cycle is equivalent to the cost of $n(n-1)$ original edges. Thus the cycle and complete graph can have the same performance for the same cost.

We therefore want to find a topology in which the diameter of the graph increases only sublinearly with the size. In fact, this is achievable even when the nodes have a constant out-degree of 2, when we use a De Bruijn graph [50]. Given a natural number, x, a De Bruijn graph has $n = 2^x$ nodes, each represented by a different bit string in $\{0,1\}^x$. To find the monitored neighbors of a node represented by $v \in \{0,1\}^x$, shift v one bit to the right, and replace the highest bit (which was shifted into the bit string) with either 0 or 1. In other words, if we rewrite $v = v_r \parallel v_0$, where $v_r \in \{0,1\}^{x-1}$, and $v_0 \in \{0,1\}$, then the neighbors are $0 \parallel v_r$ and $1 \parallel v_r$.

The distance from any u to any v is therefore at most $x = \mathsf{log}(n)$, since in the worst case, we can still shift the bits of v into u, bit-by-bit, from the least significant to the most significant. In fact, for graphs with outdegrees of at most 2, this diameter is optimal [50].

As a note on real-life deployments, the De Bruijn graph might be difficult to coordinate, given the restrictions on the graph size and the topology. A real, distributed environment could have servers administered hierarchically, in a tree. Fortunately, the tree readily provides us a topology of the defense graph, with the diameter increasing only by a constant factor: If the monitoring edges point in both directions along the tree edges, the diameter is at most twice the depth of the tree, which, in turn is logarithmic over the size of the tree, if it is balanced.

Sample Application

As a sample application, we design a protocol that protects multiple hosts, each of which runs a ClamAV process. It not only uses a concrete failure detector as the monitor, but also introduces additional features to the monitoring network. So we will first describe the features of the monitoring network, before putting it together, and describing the whole protocol.

Graph Topology

We again use the De Brujin graph. The only difference is that where a De Brujin graph has a self loop for the nodes corresponding to all zeroes and all ones, we have the nodes monitor each other. This would reduce the average case attack time, but not the worst case.

Monitor Properties

As noted in the threat model, the specific monitor we will be using are failure detectors. A failure detector starts running the original IDS process, and sends heartbeat messages at regular intervals to a certain set of peers. Having the failure detector start the monitored process lets it receive the `SIGCHLD` signal if it has terminated. At that point, the failure detector immediately broadcasts an alert. The heartbeat messages lets its peers know that it is still running. If the peers fail to receive a heartbeat within a constant multiple of the heartbeat interval, they will broadcast an alert.

Additional Features

We add two features to enhance the monitoring graph: shuffling the graph at regular intervals, and having each monitor forward the heartbeats that it receives [1].

For the first feature, we use a round counter and a shared key for a pseudorandom function (PRF). The hosts start at the same counter, 0. Then, each time a host sends its heartbeats to its monitors, it increments the counter. If the counter is divisible by a reshuffling interval, it evaluates the PRF using the shared key as the key, and the counter and each host's identity as the inputs, and uses the outputs as the randomness for creating the new graph. Therefore, as long as the rounds are synchronized, the hosts can independently calculate the same graph without network-wide interaction. Moreover, by the definition of PRF security, an adversary cannot predict the future graph, even if it can observe the current and past graph topologies, That is because as long as the adversary does not know the PRF key, the PRF is as good as random, and since it has not been evaluated on the new round, the adversary cannot predict what the output will be. The reshuffling interval will be the diameter of the graph. That way, in case of an attack, if the attack information did not propagate to all hosts before the graph changed, it will have enough time to propagate before the next reshuffling. In other words, we multiplied the attacker's time by at most 2. Another problem the reshuffling introduces is synchronization. Normally a monitor and the monitored peer are synchronized simply due to the fact

[1]We initially tried to add another feature instead of shuffling the graph: using a specialized, secure multi-party computation protocol to secretly setup the graph, so that each host only knows the hosts it is supposed to monitor, and the hosts that monitor it. But once the graph is established, the attacker can easily observe network traffic to infer the current graph Efforts to hide network traffic, as well as other side channels, turned out to be too expensive, and would undo the cost benefit of having a sparse graph in the first place.

that the monitor has to wait for the monitored peer's heartbeat before proceeding. But the reshuffling can bring together hosts that have not had direct communication since they first setup the network. To mitigate this problem, after reshuffling, a host multiplies its own wait time by one plus the maximum outdegree of each node in the graph. This gives the monitored peer time to receive the heartbeats it needs, before continuing. The host also sends a message to its newly monitored peers, informing them of its new round. Such a message will be called a rushing message. If a host receives a rushing message and finds that its own round is behind, it will not sleep for the normal interval until it has caught up, and it will send its own monitored peers similar messages containing its own, current round. This tells the monitored peers to catch up.

The second feature we add is another property of a peer that a monitor can check: is it monitoring the peers that it is supposed to? If a monitor has not received heartbeats from the peers that it is supposed to monitor, but it still does not send an alert, then the first monitor should be considered compromised. Therefore, the first monitor should also forward the heartbeat messages it received, to prove that it is performing its duties. In order to limit the message sizes, each monitor only forwards direct heartbeats, and does not reforward those that were already forwarded. Note that authenticity is important in this case, which is why we will require that the original host signs the heartbeats that it generates.

The full protocol

To enable the aforementioned features, the protocol proceeds as follows:

1. The hosts establish connections with each other.
2. The hosts genarate a signature-verification key pair, and send each other their verification keys.
3. The hosts generate and send a string that is the length of the PRF key. The XOR of the strings forms the shared PRF key, sk [2].
4. Each host sets its round counter, r, and expected round counter, $\hat{r}$, to 0.
5. At each round each host:
    a) If the round counter is divisible by the diameter of the graph, for each party, i, calculate $\mathsf{PRF.Eval}_{\mathsf{sk}}(r, i)$, and use the values to create a permutation between the positions in the graph, and the parties. Send the newly monitored peers a rushing message containing its round.
    b) To the peers that are monitoring this host, send a signed message containing its current round, r and its identity. If the host has received heartbeat messages, also forward the heartbeat messages that it received in the last round.
    c) If r is greater than or equal to $\hat{r}$, and the host has received no early heartbeats, sleep until the elapsed time since the last sleep time (or the start of the loop) is the round interval time (the sleep time is adjusted for the time spent sending and receiving messages, and performing computations).
    d) If the child has terminated, send an alert. Note that a `SIGCHLD` signal can interrupt the sleep [62]. In that case, send the alert, and then continue sleeping for the remaining time.

[2]The channels are already encrypted, so a key sharing protocol is not necessary.

e) Wait for messages from all peers. If the host expects a heartbeat message from the peer, the wait time is the round interval plus a grace period of 1 second by default, or the default value multiplied by one plus the maximum outdegree of all nodes. Otherwise, skip the peer if there is no message available.

   i. If the host receives a rushing message containing a round that is greater than $\hat{r}$, save the new round to $\hat{r}$.

   ii. If the host received an early heartbeat message, *i.e.* a peer sent an unexpected heartbeat, do not sleep for the next round.

   iii. Send an alert if the host expected a heartbeat message, but did not receive a message from a monitored peer within the wait period, or the signature is invalid, or the round in the message is less than r, or the forwarded heartbeats have an invalid signature, have a round less than $r-1$, or come from the wrong host. Note that since the forwarding peer received the heartbeats from the previous round, the monitor determines the expected identities of the hosts according to the view of the graph of the previous round.

f) Increment r.

Simulation

To illustrate the use of our metrics at scale, we will simulate the De Bruijn component of our system, and show the near-linear increase of the required adversary power over the size of the graph, which implies that even with the restriction of constant degrees, our graph is comparable to graphs in which all nodes monitor each other.

The simulation is similar to our protocol, but simplified. Each monitor pings its monitored nodes, and expects a response, which is analogous to a heartbeat message. We do not use the extra features that are not part of the graph model, *i.e.* the shuffling and forwarding of heartbeat messages, and we keep the original De Brujin graph.

Experimental Setup

Since the networked component is the only strongly connected part, and reaches the host applications, it will be the only part we consider for our simulation, which we perform using the NS-3 network simulator [77].

The two variables whose effects we measured are the size of the graph, and the interval between pings. We kept all network parameters constant. There is no loss in this network, and the bandwidth is $100Mbps$, while the delay is $6560ns$. The timeout for replying to a ping is $50ms$, which is a small value, compared to ping intervals of at least $500ms$.

We did, however, account for the fact that even in the best network environment, it would be unrealistic to assume that all nodes started at the same time, and have perfectly synchronized ping times. Thus we randomly started each node at a uniformly chosen time within the first ping interval, which is the maximum discrepancy between the ping times of two nodes. Due to this randomness, we repeated each combination of graph size and ping interval 64 times.

The worst-case adversary was designed to take the longest time possible. It would start by stopping the node whose binary identifier is all 0s, because its distance from the node whose binary identifier is all 1s is exactly the number of digits in the identifiers, which is the diameter of the graph. Afterwards, each node would be taken down at the moment it would have issued an alert.

Expected Results

To predict the required power of the adversary, we need to predict both the work it must perform, and the time it has. We know that since the graph is strongly connected, the adversary's work is all nodes of the graph.

The attack time, on the other hand, requires an estimation. Due to the logarithmic diameter of the De Brujin graph, we expect the attack time to scale logarithmically with the graph size. On the other hand, since the detection time is directly, but probabilistically, related to the ping interval, we expect the time to vary linearly with the ping interval. In any case, if we let x be the logarithm of the number of nodes, o be the timeout interval, and p be the ping interval, the time should lie between xo and $x(o + p)$.

In fact, we expect the average value to be around $x(o + cp)$, for some factor, $c \in [0, 1]$. That is to say, the slope of the attack time over the logarithm should be $o + cp$, while the slope of the attack time over the ping interval should be xc.

Consequently, the average adversary power is at least $\frac{2^x}{x(o+p)}$, which differs from the complete graph's adversary power, $\frac{2^x}{o+p}$, by only a factor linear over x.

To quantitatively confirm these hypotheses, and find the factor, c, we performed some more focused tests. First, we tried several interval times with a fixed graph size of $2^x = 256$ nodes, performing 128 trials per interval time, and calculated the best-fit line over the points representing the individual trials. We repeated this experiment, varying the graph size, with a fixed ping interval of $p = 700ms$.

Measured Results

The average adversary power and standard deviations of the 64 repeated trials for each pair of parameters are shown in Table 4.1. As expected, the logarithmic graph in Figure 4.3 shows that for every ping interval, as the logarithm of the graph size increases, the slope increases to approach a constant, reflecting the decreasing influence of the denominator. Contrast this with a complete graph: If we plotted the logarithm of the required power, we would get $log_{10}(\frac{2^x}{o+p}) = x \times log_{10}(2) - log_{10}(o+p)$, a linear function with a fixed slope, regardless of o and p. By plotting an example of such function, we see that the slope that our measurements approach is in fact that of a complete graph.

Figure 4.4 suggests that the required adversary power is linearly related to the ping frequency, and thus the network load, with the slope increasing with the graph size. Since the adversary power is inversely related to the attack time, and the ping frequency is inversely related to the ping interval, this linear relationship suggests a linear relationship between the attack time and the ping interval, as we will show next.

Figure 4.5 shows the results for the focused experiments for a graph of 256 nodes. The estimated function of the attack time, in milliseconds, over the ping interval, is $5.01 \times p + 494.60$, so that $c = 0.626$. The y-intercept differs from the estimated intercept of $400ms$ by less than a tenth of a second, less than the minimum standard deviation for any of our measured times, which was $0.186s$ for 4 nodes, with a ping interval of $500ms$.

		Ping interval (ms)					
Graph size		500	550	600	700	800	1000
4	Average (nodes / s)	6.36	5.97	5.67	4.81	4.46	3.76
	Standard deviation (nodes / s)	2.128	2.275	1.804	1.407	2.106	2.213
8	Average (nodes / s)	7.73	7.48	6.50	5.92	4.99	4.17
	Standard deviation (nodes / s)	1.544	2.046	1.348	1.388	1.219	0.953
16	Average (nodes / s)	11.36	10.30	9.22	8.21	7.38	5.93
	Standard deviation (nodes / s)	1.824	1.545	1.373	1.226	1.056	0.839
64	Average (nodes / s)	29.42	27.19	25.31	21.36	19.00	15.99
	Standard deviation (nodes / s)	3.091	2.935	2.753	2.657	2.349	1.940
256	Average (nodes / s)	85.34	79.83	73.88	64.13	57.81	45.59
	Standard deviation (nodes / s)	6.673	5.367	6.122	3.952	5.162	3.454
1024	Average (nodes / s)	278.29	255.91	238.19	208.16	185.97	150.24
	Standard deviation (nodes / s)	17.372	15.306	15.441	12.634	12.097	10.005

Table 4.1: Averages and standard deviations of required adversary power for each pair of parameters.

In Figure 4.6, for a ping interval of $700ms$, the estimated function of the attack time, in milliseconds, over the logarithm of the graph size is $501.54 \times x - 40.78$. Given a timeout interval of $o = 50.0$, we thus calculate $c = 0.645$, which differs from the previous calculation by 0.0186. And the estimated y-intercept is also close to the origin, again differing by less than any standard deviation of the measured times.

Experiments

Now we show real experiments that indicate that the simulations are practical. We also measure practical values: the setup time for the network, and the performance effect on the in-host monitor of the previous chapter, since we are now adding a second layer of protection on top of it.

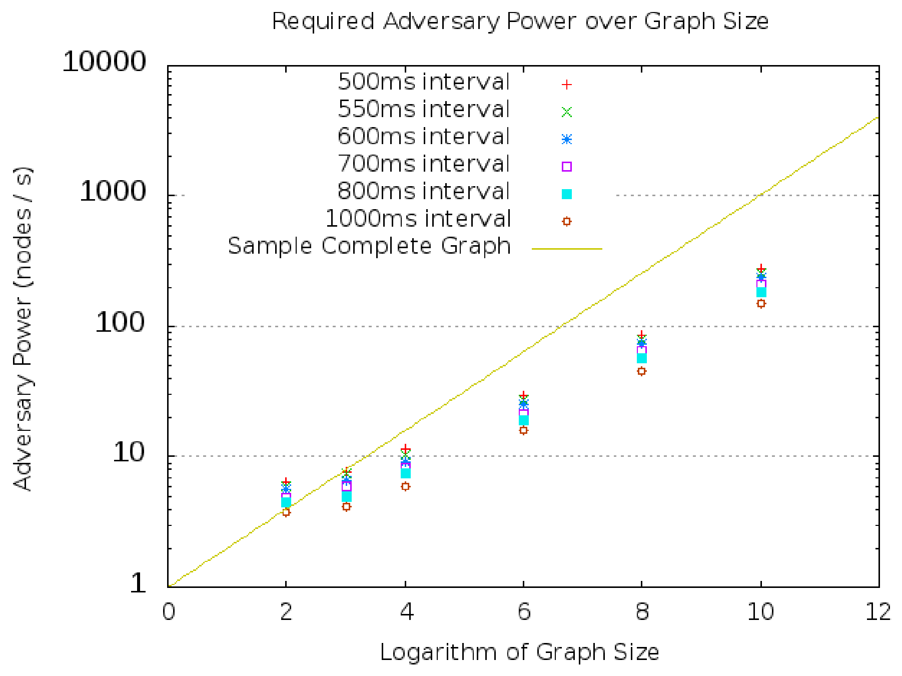


Figure 4.3: Growth of required adversary power over logarithm of graph size.

Setup

We measure detection time for two kinds of attacks: against the protected process, and against the failure detector itself. For these experiments, we use a server program to simulate attacks. If the target is the server itself, it terminates when it receives a connection; otherwise, it kills its parent, the failure detector. The attacker is a client that runs on one of the hosts that is not attacked. It measures the time between when it launched its attack, and when the failure detector on the attacker's host detected the attack.

When measuring setup time, we take into account the fact that the hosts do not start at the same time. A host retries to connect to its peers at a given interval that we will vary. In the end, we will only measure the setup time of the last host that we started, since the setup time of the other hosts includes waiting for the last host to come online.

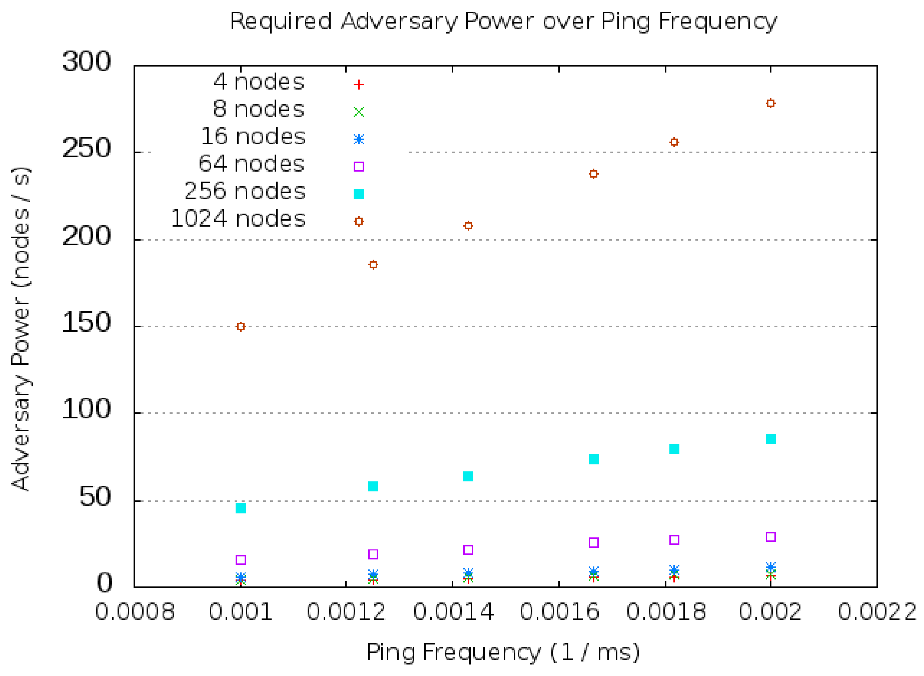


Figure 4.4: Linear relationship between required adversary power and ping frequency.

Each experiment is run for 16 trials. For the setup time measurements, we split the trials evenly between which host is started last. For the attack time measurements, we split the trials evenly between which host is attacked.

To measure the effect on the performance of the host-based monitor, we used the host-based monitor, defending `clamd`, as the protected process, and set it up in the most resource intensive setting: a heartbeat interval of 1 second and 8 hosts. Then we would measure the speed of processing files, just like in the previous chapter.

Results

The setup times are shown in Table 4.2, and with different axes in Figures 4.7 and 4.8.

The detection times for attacks against the monitored process are shown in Table 4.3, and with different axes in Figures 4.9 and 4.10.

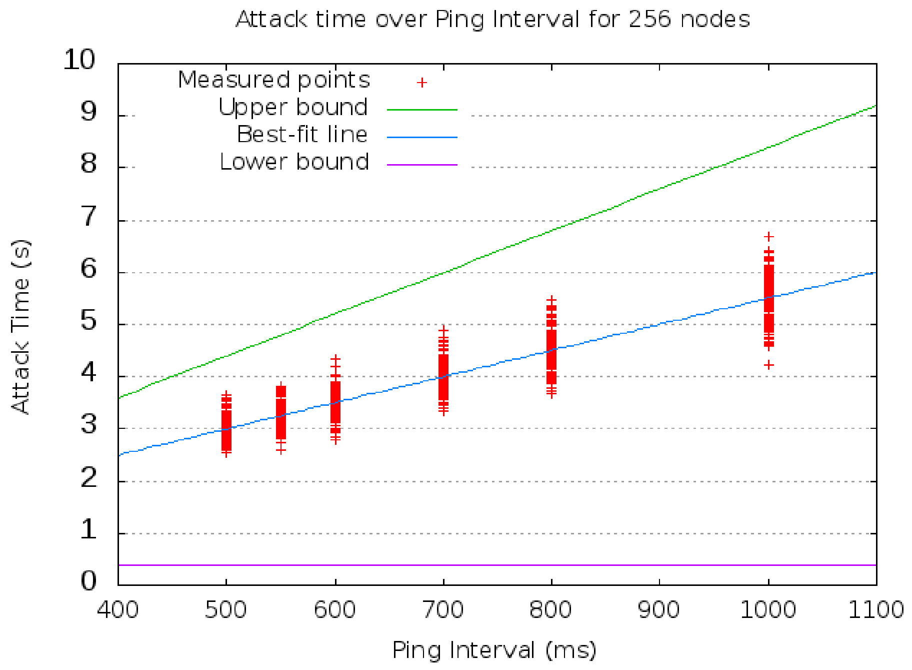


Figure 4.5: Relationship between attack time and ping interval for a fixed graph size. The estimated linear function passes through the data points. The data points all lie between the lines marking the upper and lower bounds.

Table 4.2: Graph setup times

	size (host(s))		
interval (s)	2	4	8
1.0	0.93763	1.14234	1.36153
2.0	1.93521	2.13911	2.34626
3.0	3.05957	3.13694	3.34205
4.0	3.80897	3.89528	4.33040
5.0	5.05891	4.83998	5.35277

The detection times for attacks against the failure detector are shown in Table 4.4, and with different axes in Figures 4.11 and 4.12.

As the graphs show, the times increase approximately linearly over the interval times, and vary only slightly positively over the graph size, suggesting that the network is scalable.

The additional network component has no measurable performance cost on the host-

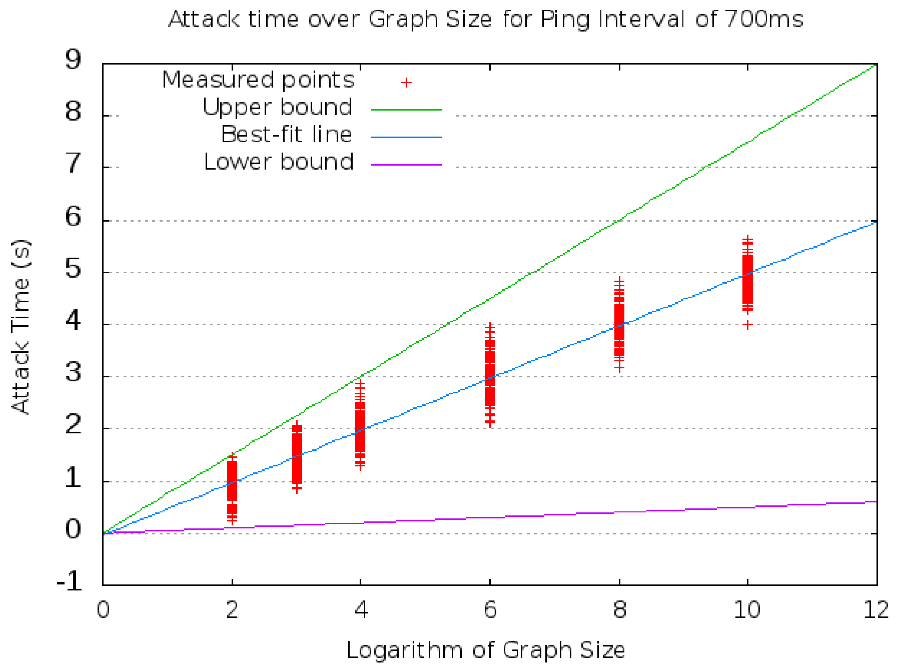


Figure 4.6: Relationship between attack time and logarithm of graph size for a fixed ping interval. The estimated linear function passes through the data points. The data points all lie between the lines marking the upper and lower bounds.

Table 4.3: Detection times for attack against the monitored process

	size (host(s))		
interval (s)	2	4	8
1.0	0.99728	0.94053	1.00428
2.0	1.62180	1.75155	1.69081
3.0	2.36957	2.12181	2.31233
4.0	2.74383	2.62158	2.93460
5.0	2.80384	2.80609	3.49662

based monitor. For the initial round, after `clamd` first started, the throughput was 109.29 files per second, which is actually an improvement over 101.16 files per second without the network. For the second round, the throughput was 303.50 files per second, an improvement over 297.92. So any performance cost is less than the margin of error.

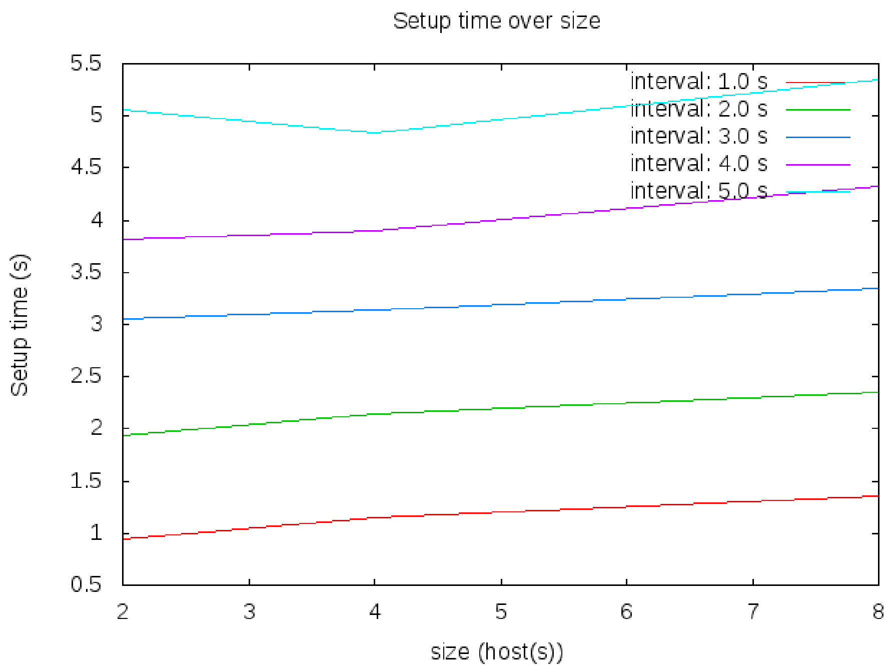


Figure 4.7: Graph setup time over network size.

Table 4.4: Detection times for attack against the failure detector

	size (host(s))		
interval (s)	2	4	8
1.0	1.56255	1.50105	1.56680
2.0	2.99784	2.93785	3.37586
3.0	3.49535	4.18614	4.93542
4.0	5.49441	5.49742	5.80895
5.0	5.05640	6.05769	7.43600

Conclusion

This chapter introduces a metric so that a network of monitors not only force the attacker to perform more work, but also perform the work quickly. Simulations and real-life evaluation show that optimizing the metric is achievable. Even though the attacker performs more work, the available time grows more slowly, thus increasing the required attack speed. Meanwhile, the implementation of the protocol for setting up and maintaining a network of attackers demonstrates that setup is inexpensive, detection is timely, and

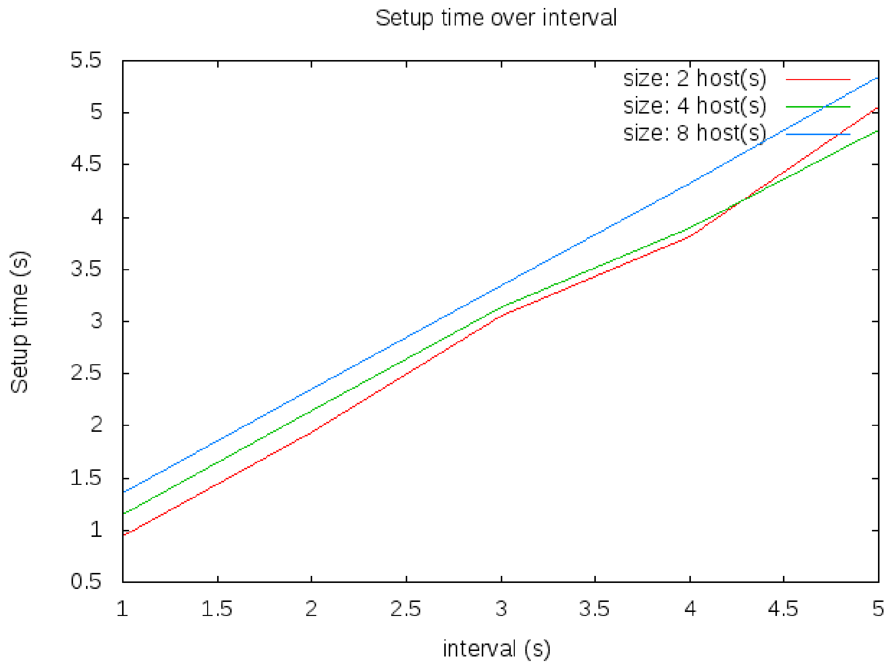


Figure 4.8: Graph setup time over retry interval.

performance cost on the protected resource is negligible.

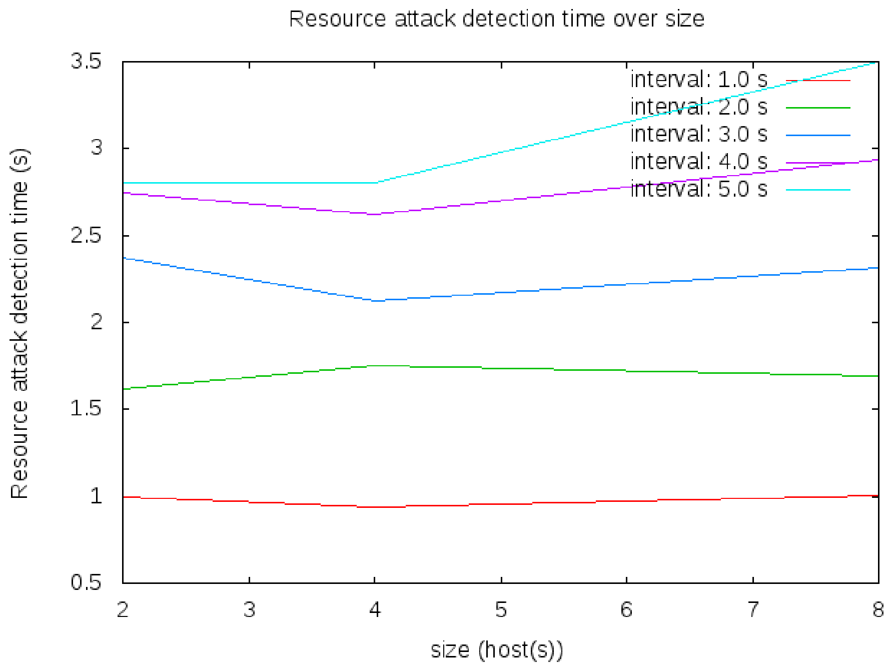


Figure 4.9: Detection time for attack against the monitored process, over network size.

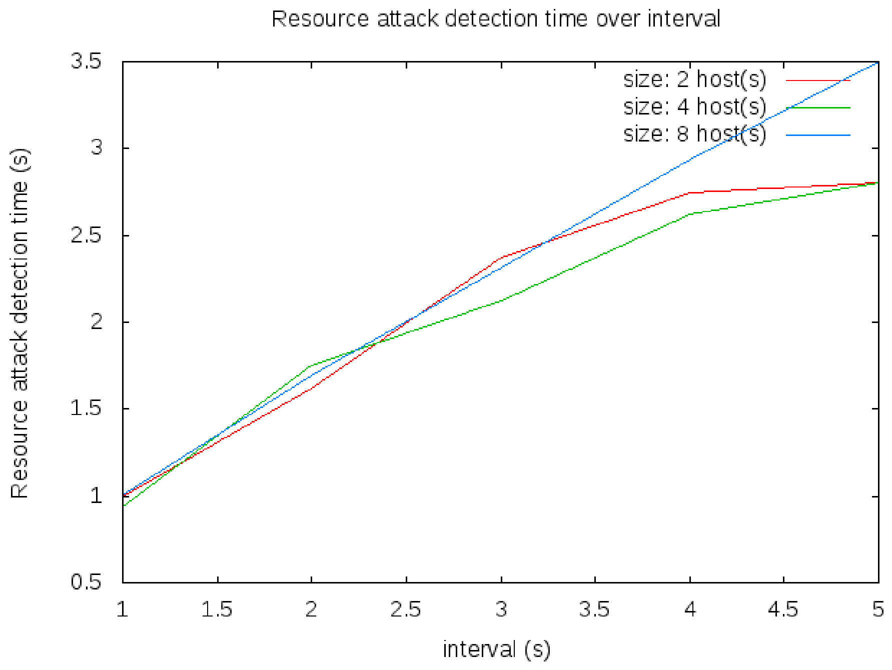


Figure 4.10: Detection time for attack against the monitored process, over heartbeat interval.

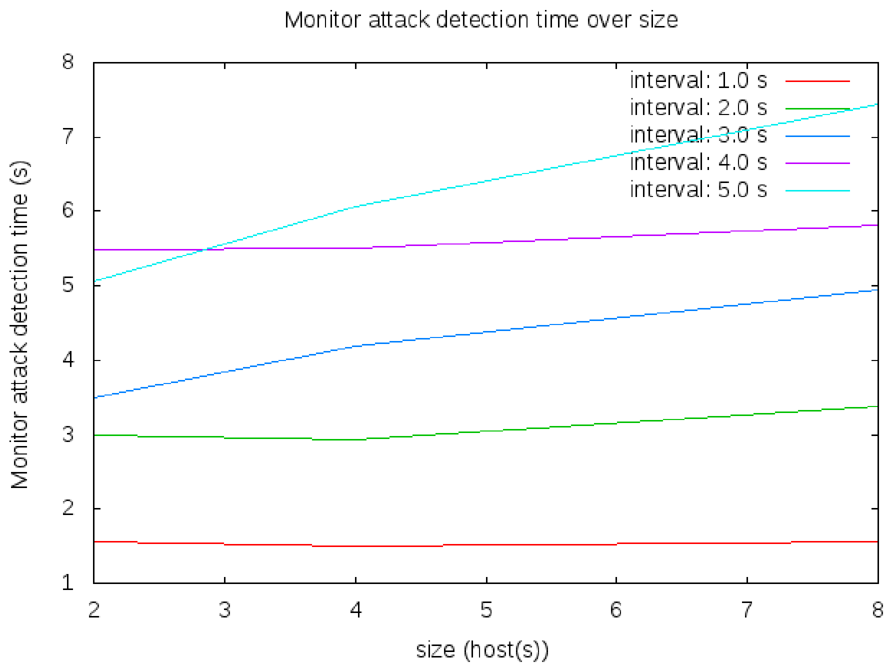


Figure 4.11: Detection time for attack against the failure detector, over network size.

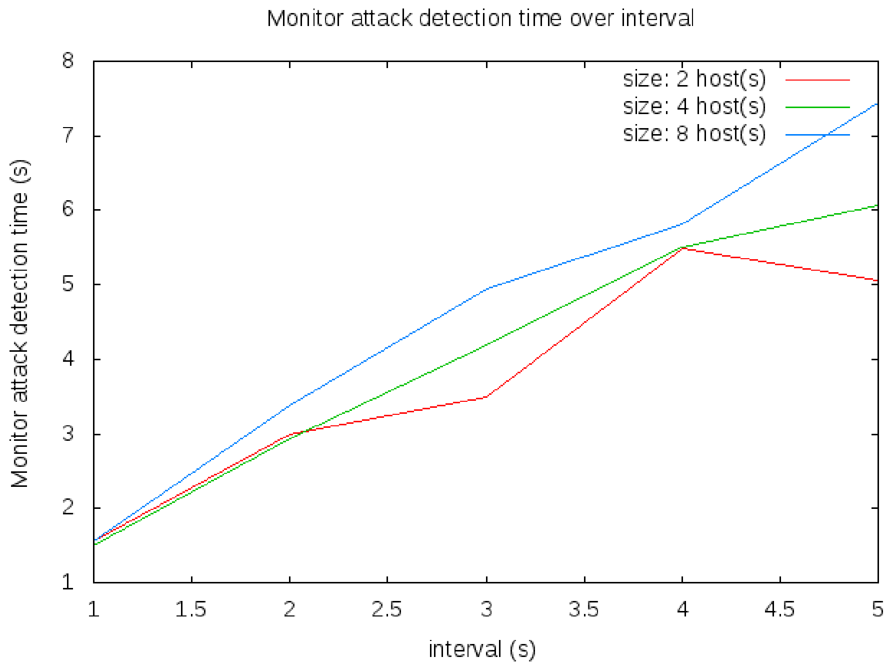


Figure 4.12: Detection time for attack against the failure detector, over heartbeat interval.

Conclusion

To make up for the shortcomings of security software, we need to strengthen or protect them with external software. But the systems and techniques described in the dissertation show that we cannot and need not perpetually add more powerful security software to protect the old ones. Often times, we can use existing programs and systems.

We can overwhelm the attacker with sheer volume, as in the case of turning bloatware into decoy applications, or forcing the adversary to attack multiple, mutually monitoring systems. Or we can manipulate the automated step of the seemingly manual process of generating fuzzing seeds to multiply the variety of seeds. This helped the fuzzer create more diverse test cases and find new bugs. If there are no available programs to help us, the new monitor we created did not need to be more complex. Instead we can focus it on a particular IDS, and tailor it towards the characteristics of IDS behavior, which turned out to be very regular, once we were able to categorize its steps.

None of these new systems require extra privileges. Fuzzing, being a preventative method, never requires extra privileges. The decoy applications only require the privileges available to a non-system application. The system call monitor needs to access the state of the IDS system it is defending, and can do so by running the IDS system. And the networked defense actually disallows unequal privileges in the cycle of monitors. As a result, we can mitigate the problem of incomplete or insecure security software, without creating additional risks.

Bibliography

[1] Available at `https://chromium.googlesource.com/breakpad/breakpad/+log/master/?s=b7b89b3b013b9b065090c2e74e04a2e6f62f5811`. (Visited on 12/16/2017).

[2] Humberto Abdelnur, Jorge Lucángeli Obes, and Olivier Festor. "Spectral Fuzzing: Evaluation & Feedback." In: (2010).

[3] *About Touch ID advanced security technology.* Available at `https://support.apple.com/en-us/HT204587`. 2017. (Visited on 12/09/2017).

[4] Anne Adams and Martina Angela Sasse. "Users are not the enemy." In: *Communications of the ACM* 42.12 (1999), pp. 40–46.

[5] *Android Debug Bridge (adb).* Available at `https://developer.android.com/studio/command-line/adb.html`. (Visited on 12/11/2017).

[6] *Apktool – A tool for reverse engineering 3rd party, closed, binary Android apps.* Available at `https://ibotpeaches.github.io/Apktool/`. 2017. (Visited on 01/11/2018).

[7] Malek Ben Salem and Salvatore J Stolfo. "Decoy document deployment for effective masquerade attack detection." In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer. 2011, pp. 35–54.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing As Markov Chain." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* CCS '16. Vienna, Austria: ACM, 2016, pp. 1032–1043. ISBN: 978-1-4503-4139-4. DOI: `10.1145/2976749.2978428`. URL: `http://doi.acm.org/10.1145/2976749.2978428`.

[9] Brian M Bowen et al. "Baiting Inside Attackers Using Decoy Documents." In: *SecureComm.* Vol. 19. Springer. 2009, pp. 51–70.

[10] J. Burnim and K. Sen. "Heuristics for Scalable Dynamic Test Generation." In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* ASE '08. Washington, DC, USA: IEEE Computer Society, 2008,

pp. 443–446. ISBN: 978-1-4244-2187-9. DOI: 10.1109/ASE.2008.69. URL: http://dx.doi.org/10.1109/ASE.2008.69.

[11] *CVE-2016-1909*. Available at https://nvd.nist.gov/vuln/detail/CVE-2016-1909#vulnDescriptionTitle. 2016. (Visited on 02/18/2018).

[12] *CVE-2016-3987*. Available at https://nvd.nist.gov/vuln/detail/CVE-2016-3987. 2016. (Visited on 02/18/2018).

[13] *CVE-2017-8390*. Available at http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8390. 2017. (Visited on 02/06/2018).

[14] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[15] Sang Kil Cha, Maverick Woo, and David Brumley. "Program-adaptive mutational fuzzing." In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 725–741.

[16] Hoi Chang and Mikhail J Atallah. "Protecting software code by guards." In: *Security and privacy in digital rights management*. Springer, 2002, pp. 160–175.

[17] Ramkumar Chinchani, S. Upadhyaya, and K. Kwiaty. "A tamper-resistant framework for unambiguous detection of attacks in user space using process monitors." In: *Information Assurance, 2003. IWIAS 2003. Proceedings. First IEEE International Workshop on*. 2003, pp. 25–34. DOI: 10.1109/IWIAS.2003.1192456.

[18] Nicos Christofides. *Graph Theory: An Algorithmic Approach*. New York, New York: Academic Press Inc., 1975, pp. 163 –165.

[19] *Cisco ASA Software IKEv1 and IKEv2 Buffer Overflow Vulnerability*. Available at https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20160210-asa-ike. 2018. (Visited on 02/18/2018).

[20] *Cisco Adaptive Security Appliance Remote Code Execution and Denial of Service Vulnerability*. Available at https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20180129-asa1. 2018. (Visited on 02/06/2018).

[21] Cloud Security Alliance. *Top Threats to Mobile Computing*. Available at https://downloads.cloudsecurityalliance.org/initiatives/mobile/top_threats_mobile_CSA.pdf. 2012. (Visited on 12/09/2017).

[22] Graham Cluley. *DarkSeoul: SophosLabs identifies malware used in South Korean internet attack*. Available at http://nakedsecurity.sophos.com/2013/03/20/south-korea-cyber-attack/. 2013. (Visited on 02/28/2014).

[23] Igino Corona, Giorgio Giacinto, and Fabio Roli. "Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues." In: *Information Sciences* 239 (2013), pp. 201–225.

[24] Heather Crawford, Karen Renaud, and Tim Storer. "A framework for continuous, transparent mobile device authentication." In: *Computers & Security* 39 (2013), pp. 127–136.

[25] *Delete or disable apps on Android*. Available at https://support.google.com/googleplay/answer/2521768?hl=en. (Visited on 12/11/2017).

[26] Dorothy E Denning. "An intrusion-detection model." In: *IEEE Transactions on software engineering* 2 (1987), pp. 222–232.

[27] *Device Administration | Android Developers*. Available at https://developer.android.com/guide/topics/admin/device-admin.html. (Visited on 12/11/2017).

[28] DoMobile Lab. *AppLock*. Available at https://play.google.com/store/apps/details?id=com.domobile.applock. (Visited on 12/10/2017).

[29] Brendan Dolan-Gavitt et al. "Virtuoso: Narrowing the semantic gap in virtual machine introspection." In: *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 297–312.

[30] Dropbox, Inc. *Dropbox*. Available at https://play.google.com/store/apps/details?id=com.dropbox.android. (Visited on 12/11/2017).

[31] Preetam Dutta. "User Behavior Analytics and User Privacy." Candidacy exam presentation. 2017.

[32] Julian Evans. *How to disable Android app Device admin rights*. Available at https://www.julianevansblog.com/2016/07/how-to-disable-android-app-device-admin-rights.html. 2016. (Visited on 12/11/2017).

[33] *Face ID*. Available at https://www.apple.com/iphone-x/#face-id. 2017. (Visited on 12/09/2017).

[34] Nicolas Falliere, Liam O Murchu, and Eric Chien. *W32.Stuxnet Dossier*. Available at http://ants.mju.ac.kr/2013Fall/w32_stuxnet_dossier(Symantec).pdf. 2011. (Visited on 01/11/2016).

[35] Henry Hanping Feng et al. "Anomaly detection using call stack information." In: *Security and Privacy, 2003. Proceedings. 2003 Symposium on*. IEEE. 2003, pp. 62–75.

[36] Stephanie Forrest et al. "A sense of self for unix processes." In: *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*. IEEE. 1996, pp. 120–128.

[37] Thomas Fox-Brewster. *Hackers Hid Backdoor In CCleaner Security App With 2 Billion Downloads —- 2.3 Million Infected.*

[38] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, California: W. H. Freeman and Company, 1979, p. 65.

[39] Tal Garfinkel and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *NDSS*. Vol. 3. 2003. 2003, pp. 191–206.

[40] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated Whitebox Fuzz Testing." In: *NDSS*. Vol. 8. 2008, pp. 151–166.

[41] Indranil Gupta, Tushar Deepak Chandra, and Germán S Goldszmidt. "On scalable and efficient distributed failure detectors." In: *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. ACM. 2001, pp. 170–179.

[42] Kenneth V. Hanford. "Automatic generation of test cases." In: *IBM Systems Journal* 9.4 (1970), pp. 242–257.

[43] Marian Harbach et al. "It's a hard lock life: A field study of smartphone (un) locking behavior and risk perception." In: *Symposium on usable privacy and security (SOUPS)*. 2014, pp. 9–11.

[44] Todd Haselton. *Credit reporting firm Equifax says data breach could potentially affect 143 million US consumers*. Available at `https://www.cnbc.com/2017/09/07/credit-reporting-firm-equifax-says-cybersecurity-incident-could-potentially-affect-143-million-us-consumers.html`. 2017. (Visited on 01/11/2016).

[45] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. "Intrusion detection using sequences of system calls." In: *Journal of computer security* 6.3 (1998), pp. 151–180.

[46] *Home | K-9 Mail*. Available at `https://k9mail.github.io/`. (Visited on 12/11/2017).

[47] *Intent | Android Developers*. Available at `https://developer.android.com/reference/android/content/Intent.html#ACTION_PACKAGE_REMOVED`. (Visited on 12/11/2017).

[48] Markus Jakobsson et al. "Implicit authentication for mobile devices." In: *Proceedings of the 4th USENIX conference on Hot topics in security*. USENIX Association. 2009, pp. 9–9.

[49] Ramaprabhu Janakiraman, Marcel Waldvogel, and Qi Zhang. "Indra: A peer-to-peer approach to network intrusion detection and prevention." In: *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*. IEEE. 2003, pp. 226–231.

[50] M Frans Kaashoek and David R Karger. "Koorde: A simple degree-optimal distributed hash table." In: *Peer-to-peer systems II*. Springer, 2003, pp. 98–107.

[51] Karspersky Lab. *How to enable/disable self-defense of Kaspersky Internet Security 2012?* Available at `http://support.kaspersky.com/6259`. 2012. (Visited on 02/02/2015).

[52] Dan Kaufman. *An Analytical Framework for Cyber Security*. Tech. rep. DTIC Document, 2011.

[53] Bogdan Korel. "Automated software test data generation." In: *IEEE Transactions on software engineering* 16.8 (1990), pp. 870–879.

[54] Joxean Koret and Elias Bachaalany. *The Antivirus Hacker's Handbook*. John Wiley & Sons, 2015.

[55] Wenke Lee, Salvatore J Stolfo, and Kui W Mok. "Mining Audit Data to Build Intrusion Detection Models." In: *KDD*. 1998, pp. 66–72.

[56] John Leyden. *Panda antivirus labels itself as malware, then borks EVERYTHING*. Available at `http://www.theregister.co.uk/2015/03/11/panda_antivirus_update_self_pwn/`. 2015. (Visited on 02/18/2016).

[57] Fudong Li et al. "Active authentication for mobile devices utilising behaviour profiling." In: *International journal of information security* 13.3 (2014), pp. 229–244.

[58] Linux Programmer's Manual. *FLOCK(2) Linux Programmer's Manual*. UNIX Manual Pages. 2014.

[59] Linux Programmer's Manual. *man elf (5)*. Linux Programmer's Manual. 2013.

[60] Linux Programmer's Manual. *man fcntl (2)*. Linux Programmer's Manual. 2015.

[61] Linux Programmer's Manual. *man lseek (2)*. Linux Programmer's Manual. 2015.

[62] Linux Programmer's Manual. *man nanosleep (2)*. Linux Programmer's Manual. 2015.

[63] Linux Programmer's Manual. *man ptrace (2)*. Linux Programmer's Manual. 2015.

[64] Linux Programmer's Manual. *open(2) Linux Programmer's Manual*. UNIX Manual Pages. 2015.

[65] *Linux System Call Table for x86 64 · Ryan A. Chapman*. (Visited on 12/16/2017).

[66] Robert Love. *Kernel Korner - Intro to inotify*. Available at `http://www.linuxjournal.com/article/8478`. 2005. (Visited on 10/22/2017).

[67] *Manifest.permission | Android Developers*. Available at `https://developer.android.com/reference/android/Manifest.permission.html`. (Visited on 12/11/2017).

[68] Timothy W Martin and Kwanwoo Jun. "'Ridiculous Mistake' Let North Korea Steal Secret U.S. War Plans." In: (Oct. 11, 2017).

[69] Patrick McDaniel. "Bloatware comes to the smartphone." In: *IEEE Security & Privacy* 10.4 (2012), pp. 85–87.

[70] Ruchika Mehresh et al. "Tamper-resistant Monitoring for Securing Multi-core Environments." In: *International Conference on Security and Management (SAM)*. 2011.

[71] Christoph C. Michael, Gary McGraw, and Michael A Schatz. "Generating software test data by evolution." In: *IEEE transactions on software engineering* 27.12 (2001), pp. 1085–1110.

[72] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities." In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: `10.1145/96267.96279`. URL: `http://doi.acm.org/10.1145/96267.96279`.

[73] Byungho Min and Vijay Varadharajan. "A novel malware for subversion of self-protection in anti-virus." In: *Software: Practice and Experience* (2015).

[74] Nader Mohamed and Imad Jawhar. "A fault tolerant wired/wireless sensor network architecture for monitoring pipeline infrastructures." In: *Sensor Technologies and Applications, 2008. SENSORCOMM'08. Second International Conference on*. IEEE. 2008, pp. 179–184.

[75] Robert Morris and Ken Thompson. “Password security: A case history.” In: *Communications of the ACM* 22.11 (1979), pp. 594–597.

[76] Leonard Mosescu. Available at `https://chromium.googlesource.com/breakpad/breakpad/+/01431c2f61aa2af1804f1e139da9bc7c4afa9e7b%5E%21/src/processor/stackwalker_amd64.cc`. 2017. (Visited on 12/16/2017).

[77] *NS-3*. Available at `http://nsnam.org`. 2014. (Visited on 01/22/2015).

[78] Notes. *ColorNote Notepad Notes*. Available at `https://play.google.com/store/apps/details?id=com.socialnmobile.dictapps.notepad.color.note`. (Visited on 12/11/2017).

[79] Tavis Ormandy. *FireEye Exploitation: Project Zero's Vulnerability of the Beast*. Available at `https://googleprojectzero.blogspot.com.au/2015/12/fireeye-exploitation-project-zeros.html`. 2015. (Visited on 02/18/2016).

[80] Tavis Ormandy. *How to Compromise the Enterprise Endpoint*. Available at `https://googleprojectzero.blogspot.com/2016/06/how-to-compromise-enterprise-endpoint.html`. 2016. (Visited on 02/18/2016).

[81] Tavis Ormandy. *Kaspersky: Mo Unpackers, Mo Problems*. Available at `https://googleprojectzero.blogspot.com.au/2015/09/kaspersky-mo-unpackers-mo-problems.html`. 2015. (Visited on 11/29/2016).

[82] Tavis Ormandy. *Sophail: Applied attacks against Sophos Antivirus*. Available at `http://dl.packetstormsecurity.net/papers/virus/sophailv2.pdf`.

[83] Tavis Ormandy. *TrendMicro node.js HTTP server listening on localhost can execute commands*. Available at `https://bugs.chromium.org/p/project-zero/issues/detail?id=693&redir=1`. 2016. (Visited on 02/18/2016).

[84] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. “Test-data generation using genetic algorithms.” In: *Software Testing Verification and Reliability* 9.4 (1999), pp. 263–282.

[85] Vern Paxson. “Bro: a system for detecting network intruders in real-time.” In: *Computer networks* 31.23 (1999), pp. 2435–2463.

[86] Nicole Perloth and Scott Shane. “How Israel Caught Russian Hackers Scouring the World for U.S. Secrets.” In: (Oct. 10, 2017).

[87] Adrian Perrig, John Stankovic, and David Wagner. “Security in wireless sensor networks.” In: *Communications of the ACM* 47.6 (2004), pp. 53–57.

[88] Brandon Perry. *american fuzzy lop*. Available at `https://foxglovesecurity.com/2016/06/13/finding-pearls-fuzzing-clamav/`. 2016. (Visited on 10/22/2017).

[89] Jonas Pfoh, Christian Schneider, and Claudia Eckert. "Nitro: Hardware-based system call tracing for virtual machines." In: *International Workshop on Security*. Springer. 2011, pp. 96–112.

[90] Alexandre Rebert et al. "Optimizing seed selection for fuzzing." In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 861–875.

[91] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C." In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: ACM, 2005, pp. 263–272. ISBN: 1-59593-014-0. DOI: `10.1145/1081706.1081750`. URL: `http://doi.acm.org/10.1145/1081706.1081750`.

[92] Ilya Shabanov. *Antivirus Self-Protection Test (August 2007)*. Available at `http://www.anti-malware-test.com/test-results/Antivirus_Self_Protection_Test_2007`. 2007. (Visited on 02/02/2015).

[93] *Snort – Network Intrusion Detection & Prevention System*. 2017. (Visited on 12/10/2017).

[94] *Source-based Code Coverage – Clang 6 documentation*. Available at `http://clang.llvm.org/docs/SourceBasedCodeCoverage.html`. 2017. (Visited on 10/22/2017).

[95] Sherri Sparks et al. "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting." In: *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE. 2007, pp. 477–486.

[96] *Storage Options | Android Developers*. Available at `https://developer.android.com/guide/topics/data/data-storage.html`. (Visited on 12/10/2017).

[97] Symantec Corporation. *Symantec Endpoint Protection (SEP) 11.0: Configuring the SEP Client for Self-Protection*. Available at `http://www.symantec.com/connect/sites/default/files/SEP_Protecting_SEP_Client_rev1.0.pdf`. 2011. (Visited on 02/02/2015).

[98] Kymie MC Tan, Kevin S Killourhy, and Roy A Maxion. "Undermining an anomaly-based intrusion detection system using common exploits." In: *Recent Advances in Intrusion Detection*. Springer. 2002, pp. 54–73.

[99] Kymie Tan, John McHugh, and Kevin Killourhy. "Hiding intrusions: From the abnormal to the normal and beyond." In: *Information Hiding*. Springer. 2003, pp. 1–17.

[100] *Testing Your App's Accessibility | Android Developers*. Available at `https://developer.android.com/training/accessibility/testing.html`. (Visited on 12/11/2017).

[101] Jonathan Voris et al. "Bait and snitch: Defending computer systems with decoys." In: *Proceedings of the cyber infrastructure protection conference, Strategic Studies Institute, September*. 2013.

[102] Jonathan Voris et al. "Secure Mobile Access using Deception." Unpublished manuscript.

[103] *Vulnerability in Fortinet FortiOS Could Allow Unauthorized Remote Access*. Available at `https://www.cisecurity.org/advisory/vulnerability-in-fortinet-fortios-could-allow-unauthorized-remote-access/`. 2016. (Visited on 02/18/2016).

[104] David Wagner and Drew Dean. "Intrusion detection via static analysis." In: *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE. 2001, pp. 156–168.

[105] Mark N. Wegman and F. Kenneth Zadeck. "Constant Propagation with Conditional Branches." In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991), pp. 181–210. ISSN: 0164-0925. DOI: `10.1145/103135.103136`. URL: `http://doi.acm.org/10.1145/103135.103136`.

[106] Alex Wheeler and Neel Mehta. *Owning Anti-Virus: Weaknesses in a Critical Security Component*. Available at `https://www.youtube.com/watch?v=BdQT99YPdJc`. 2013. (Visited on 12/08/2017).

[107] J Woodward. "Information assurance through defense in depth." In: *Directive from the Director for Command, Control, Communications and Computer Systems (J6), Joint Chiefs of Staff.(Washington, DC: US Department of Defense)* (2000).

[108] Feng Xue. "Attacking antivirus." In: *Black Hat Europe Conference*. 2008. (Visited on 03/13/2015).

[109] Yi Yang et al. "Distributed software-based attestation for node compromise detection in sensor networks." In: *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*. IEEE. 2007, pp. 219–230.

[110] Michał Zalewski. *Technical "whitepaper" for afl-fuzz*. Available at `http://lcamtuf.coredump.cx/afl/technical_details.txt`. (Visited on 11/21/2016).

[111] Michał Zalewski. *american fuzzy lop, 2.49b*. Available at `http://lcamtuf.coredump.cx/afl/releases/afl-2.49b.tgz`. 2017. (Visited on 10/21/2017).

[112] Michał Zalewski. *american fuzzy lop*. Available at `http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz`. 2017. (Visited on 10/20/2017).

[113] frank. *Chaos Computer Club breaks Apple TouchID*. Available at `http://www.ccc.de/en/updates/2013/ccc-breaks-apple-touchid`. 2013. (Visited on 12/09/2017).

[114] *llvm::Function Class Reference*. Available at `http://llvm.org/doxygen/classllvm_1_1Function.html`. 2017. (Visited on 10/22/2017).

[115] *strace*. Available at `https://github.com/strace/strace`. 2018. (Visited on 01/07/2018).

Survey of IDS Vulnerabilities

Product	CVE	Cause	Effect
Panda	–	Misclassification	Flag self as malicious [56]
FireEye	–	Executes untrusted binary	Privilege escalation to root [79]
Cisco ASA	CVE-2016-1287	Buggy packet parsing	Arbitrary code execution [19]
Symantec Endpoint Protection	CVE-2016-2208	Buggy unpacking	Kernel memory corruption [80]
FortiOS	CVE-2016-1909 [11]	Backdoor	Remote administrative access [103]
TrendMicro	CVE-2016-3987 [12]	Vulnerable RPC interface	Arbitrary code execution [83]
CCleaner	–	Malicious backdoor	Information exfiltration [37]
PAN-OS	CVE-2017-8390	Buggy DNS parsing	Arbitrary code execution [13]
Cisco ASA	CVE-2018-0101	Buggy XML parsing	Arbitrary code execution [20]

Proof of NP-hardness of Exact Metric

Delayed Dependency is a generalization of *Multiprocessor Scheduling*, an NP-complete problem in which a set of tasks, A, each of length $l_0(a)$, must be partitioned into m_0 subsets representing processors, and all tasks must be completed by the deadline $D \in \mathbb{Z}^+$ [38].

Our transformation from *Multiprocessor Scheduling* to *Delayed Dependency*, which is illustrated in Figure 1, has the nodes, which represent the tasks, force each other to complete by the deadline:

1. A: $V = A$
2. m_0: $m = m_0$
3. l_0: $\forall v \in A : l(v) = l_0(v)$
4. D: $\forall u \in A, v \in A : d(u, v) = D$
   Note that this includes $u = v$.

The reduction takes polynomial time, as it only writes a polynomial-size output. Steps (1), (2) and (3) copy values once from the *Multiprocessor Scheduling* instance. Step (4) adds $|V|^2$ items, each of size $2\lceil \mathsf{log}(|V|) \rceil + \lceil \mathsf{log}(D) \rceil$.

We now show that the existence of a solution to *Delayed Dependency* and the existence of a solution to *Multiprocessor Scheduling* are equivalent.

Delayed Dependency to Multiprocessor Scheduling

If we have a delayed dependency schedule, we can show that because at least one task must start at time 0, all tasks must finish at time D.

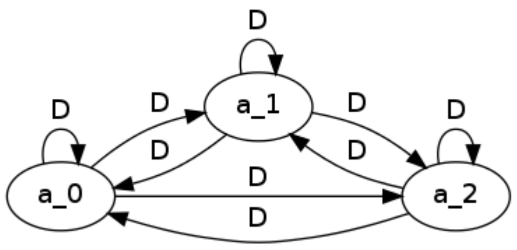


Figure 1: Illustration of reduction with 3 tasks.

Since at least one node must start at time 0, we must have a $v_0 \in V$, so that $\sigma(v_0) = 0$. Then by our definition of edges (v, v_0), each node, $v \in A$, must finish before time D:

$$c_\sigma(v) \quad \leq \quad \sigma(v_0) + d(v, v_0) = D$$

That means that all tasks have been scheduled, so $\{A_1, A_2, \ldots, A_{m_0}\}$ is a disjoint partition of all of the tasks. We now show that this partition is a valid multiprocessor schedule, *i.e.* $\forall i \in [m_0], \sum_{a \in A_i} l_0(a) = \sum_{a \in A_i} l(a) \leq D$:

Let $\{a_{i,1}, a_{i,2}, \ldots, a_{i,|A_i|}\}$ be the nodes ordered by increasing starting time:

$$\forall j \in [|A_i|], j' \in [|A_i|], \text{ so that } j < j' : \sigma(a_{i,j}) \leq \sigma(a_{i,j'})$$

Since a processor can only handle one node at a time, and the previous inequality implies that $\sigma(a_{i,j}) < c_\sigma(a_{i,j'})$, ruling out one ordering of the tasks, we must have $c_\sigma(a_{i,j}) \leq \sigma(a_{i,j'})$. Inductively, we show that $\sum_{j=1}^{j'-1} l(a_{i,j}) \leq \sigma(a_{i,j'})$:

- Base Case:
  $\sum_{j=1}^{0} l(a_{i,j}) = 0 \leq \sigma(a_{i,1})$
- Inductive Hypothesis:
  $\sum_{j=1}^{k-2} l(a_{i,j}) \leq \sigma(a_{i,k-1})$
- Inductive Step:
  Starting from our observation above:

$$\begin{aligned}
\sigma(a_{i,k}) &\geq c_\sigma(a_{i,k-1}) \\
&= \sigma(a_{i,k-1}) + l(a_{i,k-1}) \\
&\geq (\sum_{j=1}^{k-2} l(a_{i,j})) + l(a_{i,k-1}) = \sum_{j=1}^{k-1} l(a_{i,j})
\end{aligned}$$

Therefore, we can upper bound the total sum of all the lengths in a processor:

$$\begin{aligned}
\sum_{j=1}^{|A_i|} l(a_{i,j}) &= (\sum_{j=1}^{|A_i|-1} l(a_{i,j})) + l(a_{i,|A_i|}) \\
&\leq c_\sigma(a_{i,|A_i|}) \leq D
\end{aligned}$$

Multiprocessor Scheduling to Delayed Dependency

If we have a multiprocessor schedule partition, $\{A_{0,i} : i \in [m_0]\}$, we can find a delayed dependency schedule as follows:

1. A_p: $\forall p \in [m_0]$:
   $A_p = A_{0,p}$, with an arbitrary ordering, so that $A_p = \{a_{p,1}, a_{p,2}, \ldots, a_{p,|A_p|}\}$,
2. σ: $\forall p \in [m_0], i \in [|A_p|] : \sigma(a_{p,i}) = \sum_{j=1}^{i-1} l(a_{p,j})$.
   Thus $c_\sigma(a_{p,i}) = (\sum_{j=1}^{i-1} l(a_{p,j})) + l(a_{p,i}) = \sum_{j=1}^{i} l(a_{p,j})$.

This transformation achieves all three properties:

1. Start at time 0: $\forall p \in [m_0] : \sigma(a_{p,1}) = \sum_{j=1}^{0} l(a_{p,j}) = 0$.
2. At most one node per processor at a time: $\forall p \in [m_0]$ we prove this property over each task by induction:
   - Base Case: Given only $\{a_{p,1}\}$, there are no pairs that can violate this property.
   - Inductive Hypothesis: The schedule for $\{a_{p,1}, a_{p,2}, \ldots, a_{p,k-1}\}$ contains no overlaps.
   - Inductive Step: When adding $a_{p,k}$, by the inductive hypothesis, the only overlap can be due this new node. However, $\forall j < k$, $a_{p,j}$ must finish by the time $a_{p,k}$ starts:

$$\begin{aligned} c_\sigma(a_{p,j}) &= \sum_{i=1}^{j} l(a_{p,i}) \\ &\leq \sum_{i=1}^{k-1} l(a_{p,i}) = \sigma(a_{p,k}) \end{aligned}$$

3. Delayed dependency:
   $\forall v \in A$, the only deadline is due to the earliest-starting node, v_0, and an edge (v, v_0), so that $d(v, v_0) = D$. Since $\sigma(v_0) \geq 0$, it is enough that $c_\sigma(v) \leq D$, which our schedule achieves:

$$\begin{aligned} \forall p \in [m_0] \quad &, \quad i \in [|A_p|] : \\ c_\sigma(a_{p,i}) &= \sum_{j=1}^{i} l(a_{p,j}) \\ &\leq \sum_{j=1}^{|A_p|} l(a_{p,j}) = \sum_{j=1}^{|A_p|} l_0(a_{p,j}) \leq D \end{aligned}$$

NP-hardness for Constant Degree

The reduction described above generates a complete graph. However, practical defenses, including the one we designed, might have more limited in- and outdegrees. In fact, NP-completeness still holds even if the in- and outdegrees are at most 2.

Instead of creating edges between every task, we can add a new node for each task to force it to finish by time D, as described below, and illustrated in Figure 2:

Assume we have n tasks, arbitrarily ordered, so that $A = \{a_1, \ldots, a_n\}$. Construct the delayed dependency graph as follows:

1. A: $V = A \cup \{d_1, \ldots, d_n\}$
2. m_0: $m = m_0 + n$
3. l_0: $\forall v \in A : l(v) = l_0(v)$

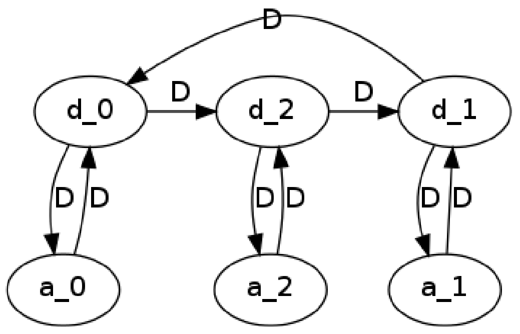


Figure 2: Illustration of constant-degree reduction.

4. D: $\forall a_i \in A$:

$$\begin{aligned} l(d_i) &= D \\ d(a_i, d_i) &= d(d_i, a_i) = d(d_{i+1 \mod n}, d_i) = D \end{aligned}$$

Thus $\forall a_i \in A : s(d_i, a_i) = 0$, and $s(d_{i+1 \mod n}, d_i) = 0$.

So in a delayed dependency schedule, if we have some d_i, so that $\sigma(d_i) = 0$, then not only must $c_\sigma(a_i) \leq D$, but, inductively, for all d_j, $\sigma(d_j) = 0$, so that $c_\sigma(a_j) \leq D$, which gives us our multiprocessor schedule. Assume this condition is not true. Then $\exists a_i$, so that $\sigma(a_i) = 0$, and $\sigma(d_i) \leq \sigma(a_i) + s(d_i, a_i) = 0$, contradicting our assumption.

Conversely, if we have a multiprocessor schedule, then we can create a delayed dependency schedule as follows:

1. $\forall d_i$, set $A_{m+i} = \{d_i\}$, with $\sigma(d_i) = 0$.
2. $\forall a_i$, schedule a_i as in the multiprocessor schedule.