THE APPLICATION OF DECEPTION TO SOFTWARE SECURITY PATCHING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Jeffrey K. Avery

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2017

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

This work is dedicated to all who have helped to make me who I am today. I thank you with all of my heart.

ACKNOWLEDGMENTS

I cannot thank my advisor, Professor Eugene H. Spafford, enough for all of his guidance, advice and support throughout my time as a student at Purdue. It is an honor and a privilege to have been taught and trained by Professor Spafford to become the research scientist I am today. The countless lessons I have learned over the years through formal and informal conversations will serve me for the rest of my life and career. I also thank my committee, Professor Saurabh Bagchi, Professor Christina Nita-Rotaru and Professor Dongyen Xu for their support and advice throughout this process.

To those who have helped me get to this point, I am forever grateful, humbled and honored. "If I have seen further, it has been by standing on the shoulders of giants." Thanks for being the giants on whose shoulders I stood.

I thank Northrop Grumman, the GEM Consortium, the Purdue Doctoral Fellowship, the VACCINE HS-STEM Career Development, and the Purdue Graduate school for their support throughout my PhD. I also thank the National Science Foundation for their support as portions of this research were funded under award number 1548114. Finally, to friends, faculty and staff in the Computer Science department, CERIAS, the College of Science, and the Purdue Graduate School, I thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SYMBOLS

T_P	Time to identify a patch has been released
T_L	Time window between when a patch has been released and when it is installed by benign end user
T_I	Time to install a patch
T_A	Time to attack
T_{PRI}	Time point at which the patch release was identified
T_{PED}	Time point at which the patch executable was downloaded
T_{VI}	Time point at which the vulnerability was identified
T_{RE}	Time to reverse engineer a patch
T_{CE}	Time to create/generate and exploit
T_{ED}	Time point at which exploit was developed
P_O	Original patch that was released to fix an actual vulnerability
P_n	Subsequent patch that has been diversified based on the original where n is a numerical value $>= 1$
λ	A security parameter

ABBREVIATIONS

SDLC	Software Development Lifecycle
SOTA	State of the Art
MTD	Moving Target Defense
OODA	Observe, Orient, Decide, Act
LoC	Lines of Code
ROP	Return Oriented Programming
PPT	Probabilistic Polynomial Time
VM	Virtual Machine
LLVM	Lower-Level Virtual Machine

ABSTRACT

Avery, Jeffrey K. Ph.D., Purdue University, August 2017. The Application of Deception to Software Security Patching. Major Professor: Eugene H. Spafford.

Deception has been used for thousands of years to influence thoughts. Comparatively, deception has been used in computing since the 1970s. Its application to security has been documented in a variety of studies and products on the market, but continues to evolve with new research and tools.

There has been limited research regarding the application of deception to software patching in non-real time systems. Developers and engineers test programs and applications before deployment, but they cannot account for every flaw that may occur during the Software Development Lifecycle (SDLC). Thus, throughout an application's lifetime, patches must be developed and distributed to improve appearance, security, and/or performance. Given a software security patch, an attacker can find the exact line(s) of vulnerable code in unpatched versions and develop an exploit without meticulously reviewing source code, thus lightening the workload to develop an attack. Applying deceptive techniques to software security patches as part of the defensive strategy can increase the workload necessary to use patches to develop exploits.

Introducing deception into security patch development makes attackers' jobs more difficult by casting doubt on the validity of the data they receive from their exploits. Software security updates that use deception to influence attackers' decision making and exploit generation are called deceptive patches. Deceptive patching techniques could include inserting fake patches, making real patches confusing, and responding falsely to requests as if the vulnerability still exists. These could increase attackers' time spent attempting to discover, exploit and validate vulnerabilities and provide defenders information about attackers' habits and targets.

This dissertation presents models, implementations, and analysis of deceptive patches to show the impact of deception on code analysis. Our implementation shows that deceptive patches do increase the workload necessary to analyze programs. The analysis of the generated models show that deceptive patches inhibit various phases of attacker's exploit generation process. Thus, we show that it is feasible to introduce deception into the software patching lifecycle to influence attacker decision making.

1. INTRODUCTION

The patching ecosystem is beneficial for end users. Software patching has the following definition: *a modification to or to modify software.*[1] An additional definition of the noun *patch* is as follows: *a collection of changed functions aggregated based on source file of their origin* [1]. These are the general definitions that form the basis for our identification of a patch.

A special case of patching is software security patches. The definition of security patch is: *a fix to a program that eliminates a vulnerability exploited by malicious hackers.*[2] An additional definition provided by Altekar et al. is *traditional method for closing known application vulnerabilities* [1].

Based on the above definitions, we use the following as the working definition for a security patch:

A modification that closes a known vulnerability in a program, eliminating the chance for that vulnerability instance to be exploited by malicious hackers.

This definition emphasizes that the modification(s) to software prohibit a vulnerability from being exploited at a specific location of a program. This does not mean that the code is hardened to all of the vulnerability instances throughout the program. Instead, the patch fixes one vulnerability at one location. Fixing a vulnerability at a specific location could hide other instances of the same vulnerability, but this side effect is not the main goal of a security patch. This dissertation will focus on both security patches and the security patching protocol.

[1]http://www.pcmag.com/encyclopedia/term/48892/patch
[2]http://www.pcmag.com/encyclopedia/term/51050/security-patch

Patches are generated once a vulnerability is identified that can be exploited. There are three paths of vulnerability discovery that lead to patch development.

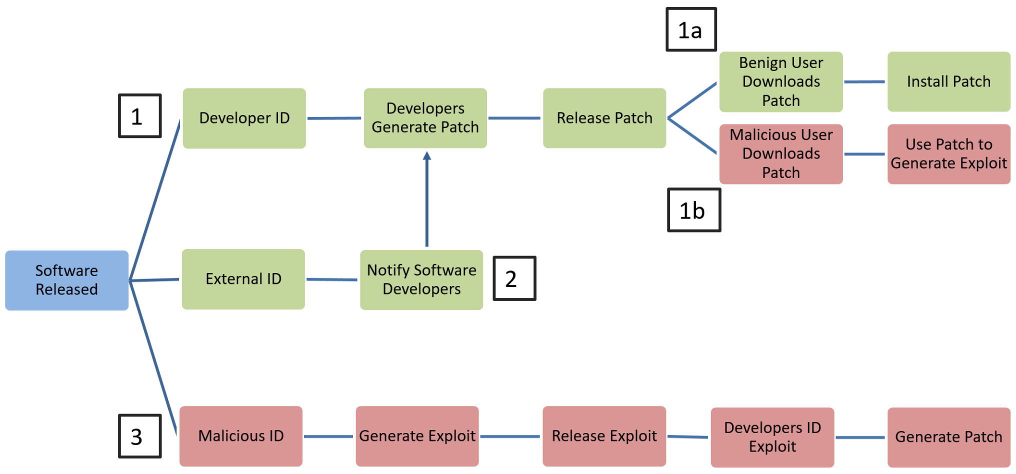

Fig. 1.1.: Vulnerability Discovery Paths

Figure 1.1 shows the various paths to identify a vulnerability in software. Path 1 is the common vulnerability path. A developer identifies a vulnerability is present in a system and develops a patch. Once the patch is released, one path is end users can download the file to their machine and install the patch. This view is the positive result of software security patching. The second path is the negative effect of software security patching. Using the released patch, an exploit can be developed. This process is called patch-based exploit generation and is the motivations for this research. The second path of vulnerability discovery begins with an external party identifying a vulnerability. These external parties notify software developers who then generate and release a patch. The third path begins with a malicious user identifying the vulnerability, generating an exploit and releasing the exploit. Developers observe attacks against their system in production and then find the vulnerability being exploited. Once this occurs, developers generate a patch and release the code to end users.

Delivery time constraints, third party programs, nonstandard coding practices, and other challenges contribute to bugs and vulnerabilities being introduced into programs that need to be fixed. The release of a software security patch (security patch or security update for short) traditionally means a vulnerability that can be exploited exists in unpatched versions of a program. This notification alerts attackers to develop exploits for unpatched systems. The potential use of patches to generate malicious exploits in practice motivates this research.

The benefit of software security patches for malicious actors is captured by a Symantec Internet security threat report [2] released in 2015 stating "...malware authors know that many people do not apply these updates and so they can exploit well-documented vulnerabilities in their attacks."

Based on this knowledge, attackers use old patches and vulnerabilities to exploit systems. This is evident by empirical research published in the 2015 Verizon Data Breach Investigations Report [3]. This report states that 99.9% of the exploits that were detected took advantage of vulnerabilities made public 1+ years prior [3]. In May 2017, unpatched systems were left vulnerable for months after a vulnerability was discovered and the subsequent patch was released, because of the lack of action by end users to apply the update. This resulted in thousands of computers worldwide, including active machines at a hospital in the UK, being compromised by the WannaCry ransomware [4]. As additional evidence of patch motivated exploits, in 2014, Tim Rains, Microsoft's Director of Security, released a blog [5] stating "[In 2010,] 42 exploits for severe vulnerabilities were first discovered in the 30 days after security updates." In June 2017, the SambaCry malware was first publicly observed, five days after the patch for the vulnerability in the Samba software package was released. As quoted from an article on the Bleeping Computer news website about the SambaCry vulnerability, “According to public data, their actions started about five days after the Samba team announced they patched CVE-2017-7494...” [3].

With time to develop exploits and the ability to access both patched and unpatched systems for testing, attackers can develop exploits that will successfully compromise vulnerable machines with high probability. Thus, traditional software security patches can assist the exploit generation process. As a result, this dissertation discusses, explores and analyzes how deception can be applied to software patching as part of the defensive strategy to enhance the resiliency of patches and help protect systems from attack.

1.1 Thesis Statement

Using deception to protect software involves prior research in obfuscation, encryption and other hiding techniques, but the specific area of deceptive patches has seen little activity. We hypothesize that:

> *It is feasible to develop a methodology to introduce deception into the software patching lifecycle to influence malicious actors' decision making and provide defenders with insight before, during and after attacks. Using this methodology, it is possible to enhance software security by using deception.*

This dissertation presents how deception can be applied to patching security vulnerabilities in software. Applying deceptive principles to the patching cycle can make attackers' jobs more difficult. The goal of deceptive patches is to increase the cost to develop exploits based on patches. These patches can cause attackers to mistrust data collected from their exploits [6], not attack a system at all to prevent wasting

[3]https://www.bleepingcomputer.com/news/security/linux-servers-hijacked-to-mine-cryptocurrency-via-sambacry-vulnerability/

resources, fear being exposed, and waste time attempting to develop an exploit for an incorrectly identified vulnerability. Thus, the impact of deceptive patches on program security is based on altering an attacker's approach, causing him/her to cast doubt on the data collected or to increase the required workload to develop an exploit. To enhance the resiliency of patches, we apply deception and discuss its impact on the workload required for attackers to generate exploits based on patches.

1.2 Patch Exploit Overview

We assume that attackers have remote access to vulnerable machines or direct access to binary or source code. We also assume varying levels of awareness to deceptive techniques. Attacks can take the form of scripted exploits, where a malicious actor has created an automated script to compromise a machine, or manual attacks.

This dissertation will focus on patches that fix security vulnerabilities. These types of patches attempt to correct flaws that have been discovered through internal review or from outside reports. In general, all vulnerabilities must be inaccessible for the system to be secure. Thus, during the design stage of patch development, the main requirement is to remove the vulnerability to prevent it from being exploited. While this requirement is enough to lead to a patch that prevents exploits from succeeding, more can be done to further secure the system using the same or similar software. This dissertation shows the feasibility of adding additional steps to the design, implementation and release stage where developers explore and potentially use deception in the process of addressing a vulnerability. Such an approach will lead to well-planned, deceptive security patches that can increase the difficulty to develop exploits, influence an attacker's decision making and expose an attacker's exploits. Adversaries targeting a deceptive patch with an exploit can inform defenders of new attack techniques once their exploit is executed. Defenders can use this information to bolster their front line preventative defense techniques proactively instead of through post-analysis after a successful exploit. Deceptive patching techniques along with traditional preventative defense techniques can help to enhance the security of software.

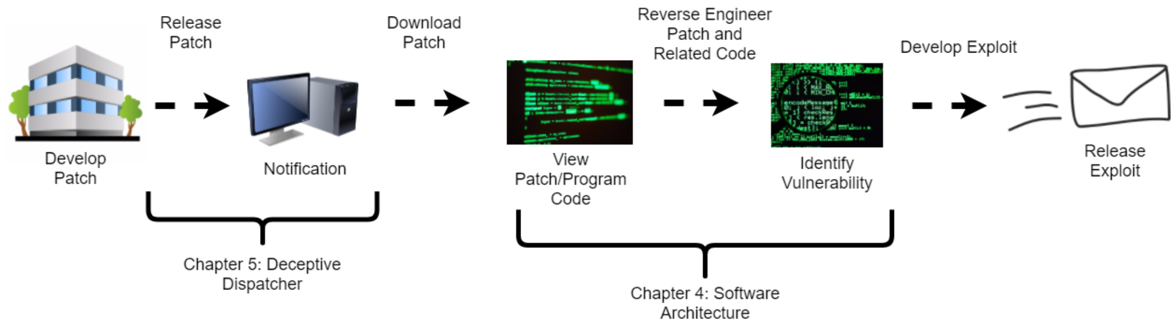

Fig. 1.2.: Patch-based Exploit Generation Timeline with Deceptive Patch Components/Research Overlay.

Figure 1.2 illustrates the patch-based exploit generation process and overlays where the concepts presented in this dissertation impact the attack sequence. The process of using patches to generate exploits begins when developers release the notification for a patch or the notification for a vulnerability (in some instances these steps are combined). Once the patch is made available to the public, an attacker reverse engineers the patch to discover the vulnerability being fixed. Once this is identified, an exploit can be developed that compromises vulnerable machines. We apply deception to security patches to slow down and/or inhibit patch-based exploit generation.

This work presents research on how deception can be applied to security patches. An outline of the contributions of this work is as follows:

1. *Explore the ability of releasing ghost patches for faux vulnerabilities to deceive attackers.*

The first contribution examines at automatically inserting fake patches into code using a compiler. Exploring techniques, such as symbolic execution, that attackers can use to develop exploits using patched and unpatched binaries can aid in the development of fake patches that appear real. These patches can mislead an attacker, causing him/her to spend extra time investigating fake vulnerabilities. This will provide end users more time to apply patches and protect their systems.

2. *Discuss a protocol update/framework using current software update centers to re-release diversified versions of patches to deceive attackers.*

   We introduce a series of steps to inject deception into the security patching process. Our analysis of inserting deceptive patches into the development and maintenance lifecycle of a program is a preliminary application of deception to the Software Development Lifecycle (SDLC).

3. *Develop and analyze a formal security model of deceptive patches.*

   We introduce a general method using game theory models to capture the security of deceptive patches. These models analyze how secure a deceptive patch is given a knowledgeable adversary and an oracle. We apply this generic model to specific instances of deceptive patches and discuss the security implications.

1.3 Dissertation Order

We discuss the outline of the dissertation and provide a brief overview of the chapters in this section.

Chapter 2 covers the background for and related work to this dissertation. We discuss a working definition of patching and prior work on the economics of patching and patching techniques. We explore how software is exploited and discuss the general area of deception and how deception has been applied to software. Finally, we discuss prior work in deceptive patching and how elements of this dissertation address limitations in these approaches.

Chapter 3 presents a model of software security patches. We discuss four components that make up a patch as well as how to apply deception to each component. We present an economic analysis of patches and deceptive patches using time to qualify the impact of deception on exploit generation. We describe a mapping of our deceptive patching model onto the cyber kill chain [7] to show how deceptive patching can affect an attacker's path to compromise.

Chapter 4 discusses what makes up a patch from the architectural standpoint. We identify components that can be visualized from a static analysis standpoint, discuss how real elements can be dissimulated or hidden and how false elements can be shown. Components of our approach to show false elements appear in the 32nd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC 2017) [8].

Chapter 5 identifies the location of a patch as a major component of the patching model. We discuss different types of patches based on where they are located in the cyber ecosystem (i.e. machines and networks). We then discuss how moving target defense (MTD) can be applied to software security patches and how the application is intuitive and a one-off approach. We also approach patching from the notification and presentation standpoint. We discuss the text within notifications that identify the presence of a patch as well as the notifications that appear during the installation of a patch. We briefly discuss how bias is exploited by deceptive operations. Finally, we describe a framework that given a patch, diversified versions of the patch can be released after the original patch, forcing an attacker to distinguish between an original patch and a diversified version of the same patch to avoid attempting to exploit a vulnerability that has a previously released patch, expending his/her resources.

We suggest the need to distinguish between diversified versions of the same patch will increase the workload required for attackers to develop patch-based exploits and discuss how this can be applied to a generalization of the current software security patching protocol using existing research. Elements of this chapter can be found in 12th International Conference on Cyber Warfare and Security (ICCWS 2017) [9].

Chapter 6 discusses a game-theoretic approach to describing the resiliency of a deceptive patch. We discuss this general approach and then provide applications to different categories of deceptive patches.

Finally, Chapter 7 concludes this dissertation and provides direction for future work.

2. LITERATURE REVIEW

We explore the literature that relates to and is background for deceptive patches. We begin by exploring the research present on software patching, reviewing the definition of a patch, types of patches, as well as economic principles that support patching and patch development. We also discuss how patches can be exploited, which significantly motivates this research and identify related work on deceptive patches.

There are four ways a patch can alter code: add new lines of code at the site of the vulnerability, change or edit vulnerable lines of code, remove vulnerable lines of code, or wrap the vulnerability in a protective block. Adding, editing, and removing vulnerable lines of code operate internally to a susceptible function at the site of a vulnerability. These modifications prevent exploits from succeeding at the site of the vulnerability by detecting and/or addressing unauthorized changes in local state variables or removing the flawed code. Wrappers operate external to a vulnerable function. Wrappers can either cleanse input to a function before it is used or examine output of a function to verify its correctness before it is used in other locations throughout the program. Wrappers can detect exploits if the exploit alters the system's state and/or program's behavior.

2.1 Types of Patches

Patches can be categorized based on the developer, its application to the code, length, as well as what elements they actually update. Patches categorized based on the developer can be unofficial (those developed by 3rd party vendors) or traditional (those developed by the original code developers) [10]. While patches may require a system restart to be applied, hot patches are applied to software as it is executing without the need to restart. For example, prior work by Payer et al. explores how dynamic analysis and sandboxes can provide patches for vulnerabilities during runtime [11]. Hot patches provide one solution to inhibiting patch-based exploit generation, but are not a general solution. For example, hot patches could cause instability on a machine because of compatibility issues with existing programs running on a machine, which could be unacceptable for end users. Patches can also be categorized by length, either in lines of code or memory size. Using length as a delimiting factor can help identify the amount of code necessary to fix classes of vulnerabilities. Longer patches that change large sections of code are called bulky patches or service packs, while patches that change small portions of code are called point releases [10]. Security patches and data patches are based on the software element(s) they update. Security patches update vulnerable software components of a program that could be exploited, and data patches update rules and signatures used by protection applications to detect attacks [12]. We focus on security patches that use a traditional update mechanism, though our approach can be applied to any mechanism.

Finally, patches can be categorized based on their location relative to the location of the vulnerability being fixed. External, or wrapper, patches are implemented in a separate location compared to where the vulnerability is located. For example, a buffer overflow attack where the variable's size is known prior to entering the function can be detected by an external patch. Internal patches are located inside a vulnerable function and address the vulnerability by adding, editing and/or removing code within the function. This type of patch can detect and prevent exploits as soon as they occur, taking the necessary action(s) in real time. This allows for exploits to be detected in dynamic environments where variable sizes and locations are non-deterministic. Internal patches also have access to the internal state of a function. Zamboni et al.

provide a complete analysis of internal and external sensors, of which patches are a subset [13].

2.2 Patch Development Lifecycle

Over the lifetime of an application, developers continue to update code, find vulnerabilities, discover areas where the code can be optimized, or add new features. Updating code should follow a series of steps, ensuring the patch performs its intended functionality and does not add incompatibilities. Patches either fix vulnerabilities in code or aesthetically improve older versions of code. Brykczynski et al. describe a series of sequential steps to develop a security patch [14]. Figure 2.1 diagrams a general patch release process and each tier is described in Table 2.1 [15].

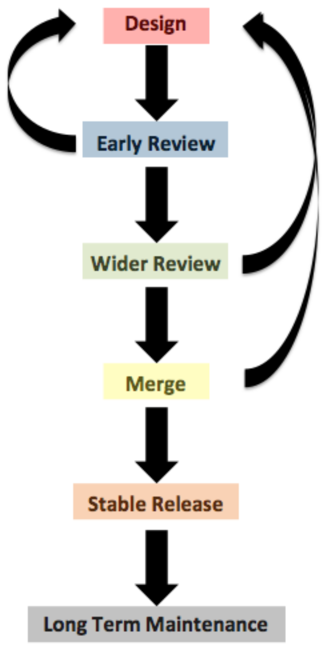

Fig. 2.1.: Lifecycle of patches

Table 2.1.: The Patching Cycle

Design	Develop patch requirements - usually done without community involvement
Early Review	Post patch to relevant mailing list; address any comments that may arise; if there are major issues, developers return to the design stage
Wider Review	More extensive review by others not involved in the early review; if there are major issues, developers return to the design stage
Merge	Place patch into mainline repository
Stable Release	Deploy patch to the public
Long Term Maintenance	Developers maintain the patch as the code undergoes other improvements

The traditional patch lifecycle also shows that there are multiple stages of review and testing that take place to make sure the patch is suitable to fix the vulnerability. Vendors want to make sure that the issue is completely fixed and confidently ensure that additional issues with that vulnerability do not arise.

The patch development lifecycle models the major stages to fixing vulnerabilities in code. The original image presents a waterfall type of model where each stage leads into the next upon completion. We slightly alter this model, adding additional feedback loops, representing a more granular approach to patch development.

This lifecycle suggests that there exists an expectation that a patch fixes an issue present in code. This also suggests that the issue is a vulnerability present in the code that can be exploited. If a vendor is going to spend time reviewing, testing, and fixing one of their mistakes, the fix for the mistake should be correct in the sense that it actually fixes the error in the code. This belief that security patches always attempt to fix legitimate vulnerabilities supports the application of deception.

Specifically, deception is applied to the design and merge stage of the software patching lifecycle. The accepted belief is a patch fixes a vulnerability that is exploitable in the software. Fake patches are one way to apply deception to security patching to take advantage of this expectation that a patch is always code that addresses a real vulnerability. One challenge of adding fake patches is these patches cannot alter data flow or control flow in such a way that the program performs unreliably for benign and legitimate use. We address the idea of fake patches in Chapter 4.

Deception is also applied to software security patching during the stable release stage. This can be achieved by adding deceptive notifications and releasing patches that are diversified versions of prior updates. We discuss this in more detail in Chapter 5.

2.3 Patching Economics

The ecosystem of software development involves economic trade-offs between releasing an application and further developing software [16]. Economic principles guide when and how software is updated and when these updates are released. Time to fix a bug, delivery vehicle, and vulnerability criticality all contribute to patch economics. At its core, patching software is a risk management exercise [17, 18]. Identifying the risks and rewards associated with a security patch helps guide developers as they decide when to release updates. Managing this process and decisions that are involved in a practical setting are discussed by Dadzie [19] and McKusick [20].

The economic culture of patching suggests that patches are released within optimal windows of time after a vulnerability has been identified or an exploit has been publicly released. This means that patches are released when a significant amount of data about the vulnerability and corresponding fix have been gathered as well as a minimal amount of time has passed since public notification. Studies also suggest that public notification of a vulnerability increases the speed to patch a program [18, 21].

The current software development phase of an application impacts the economics of a patch. If a patch is identified during testing, applying the patch could be more economically efficient when compared to releasing a patch when software is in full production mode. Finding bugs when code is in the maintenance phase costs more than finding them in the production or development phase [16].

2.4 Patch Generation

Patches can be generated using manual analysis and coding or automated tools. We briefly discuss prior work that studies manually and automatically creating patches.

2.4.1 Manual Patch Generation

Manual patch generation identifies vulnerabilities to be fixed using manual effort. Once identified, the patch for the vulnerability is written, tested, and released by developers [22]. A full treatment of this type of generation is outside the scope of this work. Research by Sohn et al. explores improving manual patch generation for input validation vulnerabilities [23].

2.4.2 Automated Patch Generation

A growing area of research uses static and dynamic analysis techniques to automatically find and patch vulnerabilities. An overview of software repair concepts is provided by Monperrus [24]. Research by Wang et al. detects integer overflow vulnerabilities and provides a patch to fix the flaw [25].

Deception can also be applied to influence the information provided by these tools. We apply this concept by generating faux patches for input validation vulnerabilities by inserting fake conditional statements that model actual patches. This is discussed in more detail in Chapter 4.

2.5 Software Exploit

Hackers exploit publicly available applications by forcing the program to perform functions that were not intended. One of the first steps to altering program behavior is gaining an understanding of how the software operates. To achieve understanding, attackers apply reverse engineering techniques to provide human readable analysis of the application.

2.5.1 Vulnerability Research

Identifying and classifying vulnerabilities based on how they are introduced to code can be used to develop more secure coding practices. Prior work by by Jang et al. explores finding vulnerabilities based on prior patches for a given application [26]. Work by Krsul provides a full treatment of vulnerability analysis and categorization [27]. Work by Xie et al. uses static analysis techniques to identify vulnerabilities in software [28]. Analysis by Frei et al. uses patch and exploit release data to identify trends in vulnerability detection, exploit, and patching [29]. Deceptive patch development relies on vulnerability research to identify classes of vulnerabilities.

2.5.2 Application Exploit

Attackers use exploits to attack vulnerabilities in unpatched applications. These attacks can provide attackers the ability to gain access to otherwise unavailable functionality. One of the first steps in this process is reverse engineering the code to either view its contents or identify vulnerable patterns in the program. Research has identified major questions within reverse engineering and provided advances within the field. This research also increases the scope of code and programs that can be reverse engineered. Work by Rugaber et al. attempts to qualify the accuracy and completeness of a reverse engineered program [30]. Research by Schwarz et al. and Popa looks at reverse engineering executables and binary with non-standard coding practices such as indirect jumps and code vs data identification [31, 32]. Udupa et al. study how to reverse engineer programs deceptively treated with obfuscation techniques [33]. Prior work by Wang et al. looks at defeating these attack techniques by reviewing how techniques such as encryption, anti-debugging code and even obfuscation can increase the difficulty to reverse engineer applications [34]. This limited availability of "anti-reverse engineering" techniques is where deception can be applied. Adding fake patches does not prevent reverse engineering from occurring, but it does alter the data to be analyzed, increasing the workload of an attacker.

2.5.3 Patch-Based Exploit Generation

Attackers use released patches to develop exploits against unpatched machines [35–38]. This is possible because software patches inherently leak information about the software being updated.

Binary Diff One disadvantage is that internal patches leak the location of a vulnerability that is present in unpatched code, providing attackers with a blueprint to develop exploits against unpatched code that can be verified. A binary difference (or *diff*) reports the differences in syntax between two provided binary files [39, 40]. The *diff* result can then be used to start the reverse engineering process and the exact lines of code that were changed can be observed. This provides attackers with the same patch that has been distributed to all other users of the application and as a result, because of the patching monoculture where all systems receive the same update, the vulnerability on all unpatched systems can be identified. This static analysis process can be used to develop exploits manually [41,42].

Deception can affect binary diff tools by adding deceptive patches to code that increases the size of the diff result. By adding fake code to a patch, the amount of information returned as a result of executing the *diff* command on a deceptively patched and unpatched system could increase the workload or be too large to analyze. Chapter 4 explores the impact of injecting deceptive patches into software.

Control Flow Analysis Another disadvantage is patches alter the behavior of a program. Once the patch is applied, the result is a more secure and hardened program, but in the timeframe between patch release and patch installation, this observable difference is harmful. This altered behavior, when its outputs are compared to an unpatched system over a range of inputs, can be used to identify the functionality of a patch, and therefore the vulnerability being fixed. Because the behavior is altered, the search space for an attacker using fuzzing, which as a technique is similar to brute forcing a program to attempt to make it behave erratically, is diminished as a change in program behavior can be used as the initial identifier that a patch is present and can help identify inputs that trigger the execution of patch code [43,44].

Attackers can use control flow analysis to identify a vulnerability based on its patch. Analyzing, statically or dynamically, changes in the control flow graph between a patched and unpatched system can expose the location, syntax and/or behavior of a patch. This provides attackers with information about the vulnerability being fixed.

Deception can impact control flow analysis by increasing the number of paths in a program or by hiding distinct paths in a program. Control flow obfuscation techniques such as control flow flattening cause the control flow of a program to be more difficult to identify and follow statically. Chapter 4 discusses how the application of deceptive patches can alter control flow and increase attacker workload to develop exploits based on patches.

Symbolic Execution A more efficient fuzzing technique is symbolic execution, which uses symbolic or representative input values based on conditionals in code to enumerate the paths throughout a program [45]. This technique can also be used to dynamically identify new paths that are executed during runtime between a patched and unpatched program. Identifying the different paths throughout a program and new paths could expose the behavior of a patch and provide information about input values necessary to exploit a vulnerability [38]. We apply symbolic execution to program analysis as an indication of the workload required by an adversary performing program analysis. Deception can increase this workload by adding fake branch statements in code. Chapter 4 explores this concept in more detail.

2.6 Deception

Deception has been used in computing since the 1970s [46–49]. Since its introduction, a variety of deceptive tools have been developed to bolster computer defenses. Examples of deceptive tools are those that generate decoy documents [50], honeyfiles [51], as well as the Deception Toolkit [52]. These documents are planted to attract attention away from critical data or resources and alert defenders of potential intrusions or exfiltration attempts. Though the negative applications of deception receive most of the focus — a phishing attack that resulted in millions of stolen credentials or malware that compromises machines across the world — benevolent applications of deception exist. An in depth analysis of benevolent deception can be found in [53, 54]. Below we present the definition of deception that we will use throughout this work. Additional work on military deception, deceptive theory and taxonomies have also been addressed, but analyzing this research is outside the scope of this dissertation [55–58].

2.6.1 Working Definition of Deception

Deception has varying definitions based on different psychological principles. The definition we will be working with is as follows: *Planned actions taken to mislead and/or confuse attackers/users and to thereby cause them to take (or not take) specific actions that aid/weaken computer-security defenses* [59, 60].

The above definition shows that an actor's *intent* to manipulate an individual's perception is the main principle of deception. In the use of deception, one party *intentionally* alters, creates or hides information to influence the behavior of other parties.

In practice, deception can be separated into two components that work in tandem. One element is hiding the real - dissimulation, and the other is showing the false - simulation. Below is a general taxonomy of deception from prior work by Bell et al. [61, 62] that we use throughout this dissertation:

1. **Dissimulation**: hiding the real
    (a) Masking
    (b) Repackaging
    (c) Dazzling
2. **Simulation**: showing the false
    (a) Mimicking

(b) Inventing

(c) Decoying

Using these components of deception, we evaluate the effectiveness of deception's application to security. Specifically, we will look at deception's application to software patching in non-real time systems.

2.6.2 Applying Deception to Software

Program Obfuscation Obfuscating code can be carried out in a variety of ways. One technique makes code difficult to understand and read by reorganizing statements or altering statements that hide a program's semantics. Another technique makes the behavior of code more difficult to understand. Introducing noise to output can make this more difficult to understand. Prior work by Collberg et al. provides a taxonomy of software obfuscation techniques [63, 64].

Software Diversity Software diversity is an area of study that researches ways to create a more diverse software base. Different versions of a program that all reach the same output using different techniques and instructions limit the reach of any one exploit developed against a vulnerability exposed in a program. This makes the attackers' task of generating an exploit with far-reaching success more difficult to accomplish because multiple versions of an exploit may have to be developed to achieve the same result of compromise. Research by Larsen et al. provides an overview of software diversification [65].

2.6.3 Deceptive Patches

Diverse Patch Applying software diversity to software security patches is a specific application of Moving Target Defense (MTD) techniques [66]. Patch diversity addresses the mono-culture problem created by current patching practices and could increase the resources needed to develop patch-based exploits. A framework presented by Coppens et al. introduces the idea of using diversification to protect patches by releasing different versions of the same patch to end users [67]. Because there could be multiple patches released for a single vulnerability, attackers must develop multiple exploits for each version of a patch to have the potential for a widespread attack [65, 68, 69]. This dissertation builds on these frameworks by showing how diversification can be realized using current patching protocols. Chapter 5 presents a framework using currently available tools to re-release diversified versions of patches.

Faux Patch A *faux* patch is composed of fake patches for vulnerabilities that do not exist in the same sense that a traditional patch is composed of legitimate patches for vulnerabilities that do exist. Fake patches should be indistinguishable from legitimate patches and force adversaries to expend resources searching for a vulnerability that does not exist. A faux patch, in combination with a traditional patch, creates a *ghost* patch. We study faux patches applied to input validation vulnerabilities in Chapter 4. Input validation vulnerabilities occur when developers do not include checks and assertions that validate data input into a program. The traditional method of fixing this type of vulnerability is to add conditional and/or assertion statements to the code that can detect invalid input [38]. Thus, we use deception to take advantage of this commonly used technique to fix this type of vulnerability. Fake patches share similarities with decoy documents [50, 51, 70, 71] and decoy passwords [72], as they are all red herring techniques [73].

Fake patches incorporate properties from legitimate patches, such as boundary checks, value verification conditional statements, and permission checks but do not alter program semantics. Prior work has suggested implementing and publicizing faux patches, but no experimentation has been conducted on this topic [42, 74]. We discuss our treatment of adding fake patches to code in Chapter 4. We develop a compiler-based implementation that adds fake conditional statements to programs and analyze the impact of the fake code. We analyze both runtime and workload impact of these faux patches on programs and present our findings.

Obfuscated Patch An *obfuscated* patch fixes a legitimate vulnerability but is ideally designed to be infeasible to reverse engineer and uncover the underlying flaw. These patches increase the effort necessary for the adversary to identify the vulnerability being fixed by the patch. Because these patches fix legitimate vulnerabilities, they do alter the semantics of the program. The goal of these patches is to confuse attackers as they develop exploits, burying the actual vulnerable code in layers of obfuscated patch code. Prior work in this area has explored code diversification [67], control flow obfuscation [63, 75], and encrypting patches [74].

Active Response Patch An *active response* patch will fix the underlying vulnerability, but will respond to adversarial interaction as if the vulnerability is still present (and potentially issue a notification of the intrusion) [76]. When interacting with an active response patch, attackers should ideally be unable to identify whether the remote system is patched or vulnerable. The main goal of these patches is to influence attackers to believe their exploit was successful. This will allow defenders to monitor the adversary's actions throughout his/her attack. Prior work has suggested these types of patches would be effective against remote attackers [77, 78].

The adversary is assumed to have access to the patch, though even with this knowledge they should be unable to achieve a meaningful advantage in differentiating between interactions with a patched and unpatched system. We show that of these three deceptive patch techniques, active response patches are the most likely to satisfy a meaningful security definition and be realized and deployed in practice.

2.7 Related Work

Work by Arujo et al. introduces the idea of a *honeypatch* [77]. A honeypatch is composed of two major parts. The first component fixes the vulnerability and the second component is a detection element that actually can detect when an attack is occurring. Thus, if malicious input is received, this input is detected as malicious and then execution is transferred to a honeypot environment that has the same state as the original machine, including the exploited vulnerability and other vulnerabilities that have been intentionally left in the honey pot. Thus, the behavior of patched compared to unpatched machines appears equivalent when in reality, the patched machine is protected against attacks exploiting the associated vulnerability [77]. Zamboni et al. similarly study how patches can raise alerts once an attack is detected [13]. The limitations with these works include the lack of automation to insert honeypatches into vulnerable code and the dependency on an attacker's inability to identify a honeypot environment. Specifically, the ability to identify honeypot environments has been shown in research [79] and during live exercises [80].

Crane et al. present a framework that describes how code can be instrumented to place fake instructions in locations where an attacker expects real instructions. These instructions would not be used by legitimate programs, but send information to defenders if they are executed [81]. The main limitation of this work is the lack of implementation and analysis that shows the feasibility of this technique in practice.

Bashar et al. discuss how patch distribution can be achieved without leaking information and providing some analysis of applying deceptive techniques as a solution [74]. This dissertation expands on concepts presented in this work by implementing a fake patch insertion compiler and providing a software security development protocol that applies software diversity to patches.

This dissertation advances the field of deceptive patching by analyzing a formal model analyzing the impact of deception on software security patching, implementing an automated fake patch compiler, and using this implementation to perform analysis on fake patch generation.

3. A MODEL OF DECEPTIVE PATCHING

This chapter presents a model of deceptive patches. This chapter examines the space of deceptive patching by first exploring the four basic elements of a software patch. This allows us to overlay deceptive principles onto the patching model to create a model of deceptive patches. Showing how the deceptive patch model interacts with the cyber kill chain attack model identifies the potential impact deceptive patches have on the stages of an attack. The cyber kill chain model is ideal for analyzing deceptive patches because it captures the attack process, including exploit generation.

3.1 Patch Components

The space of patches is categorized into four areas that have distinct properties from each other. Each category embodies a unique set of challenges and solutions for deceptive applications. This also helps to provide recommendations, suggestions and protocols for applying specific deceptive techniques to certain areas and expose areas where deception is infeasible or redundant. Our model of a patch is separated into four categories.

- Software Architecture - The physical and measurable components of a patch. This includes elements such as patch size (Lines of Code (LoC), memory size), loops, conditional statements, and variables. These elements can be measured without executing a patch.
- System Input and Output - The activity of a patch. System Input and Output captures the behavior of a patch, including input into and changes in machine

state as a result of a patch. These metrics can be observed while a program is executing before, during and after patch code executes. These metrics can also be observed if code is symbolically executed.

- System Architecture - Where the patch is located. Within the vulnerable function, in the firewall or outside the vulnerable function are all viable locations for a patch.
- Deploy and Install Chain - The public release stating a patch is available. This is how end users are notified that a patch is available. Also, the information portrayed to end users during a patch's installation is included in this chain.

These categories model the elements of a patch. These are the building blocks of a patch, and these are the elements to which deception has been and can be applied.

3.1.1 Software Architecture

Software architecture encompasses the structural make up of a patch and all attributes that can be measured from them. This includes any information that can be gathered from static analysis or running tools on the patch, whether the patch has been installed or not. Any information that can be gained without explicitly executing the patch falls in this category. This includes LoC, patch size in memory, paths in a patch, number of loops, conditional statements, number of variables, number of basic blocks, coding language and variable names.

3.1.2 System Input and Output

System input and output (System I/O) includes elements of a patch that are input into a patch, present when a patch commences execution, or output as a result of executing the patch. For example, this category consists of program runtime of a patched program, register values during patch execution and dynamic control flow during execution. System input and output also encompasses information that may be leaked during the execution of a patch. For example, a patch that prevents specific input values leaks information about the state of the program as well as the functionality of a patch [82].

3.1.3 System Architecture

System Architecture elements of a patch include where the patch may be placed. The basic idea is that a patch has to be placed somewhere within the computing ecosystem. Wrapper patches are located outside the function that is vulnerable but detect pre and post conditions [13]. Data patches are implemented in firewalls and on sensors that detect malicious traffic well before reaching any vulnerable machine [12, 83].

3.1.4 Deploy and Install Chain

Deploy chain elements of a patch include any publicly available information of a patch that can be downloaded and installed. This information can be pushed to end user machines and displayed via update applications, email, or placed on software vendor websites for users to download.

Install chain includes elements concerning installing the patch and explaining sections of a program that will be altered. Elements such as ease of understanding, ease of installing and the feedback mechanisms that are provided as a patch is installed are other parts of the install chain. In addition to the code, a patch includes information that provides notifications to system administrators and end users about the state of patch installation and files that were changed, added or removed.

The types of notifications within this category are as follows:

- Identification - information indicating that a patch will be released or has been released.
- Release - information provided at the moment a patch is released. This notification could be incorporated with the identification notice.
- Download - information displayed when a patch is downloaded onto a machine.
- Execution - information presented when a patch is installed on a machine.
- Result/Outcome - information shown regarding the successful or unsuccessfully installed on a machine.

3.2 Applying Deception to Patch Components

Each element of a patch can be deceptively influenced. We explore the basic components of applying each principle of deception to each element of a patch to serve as building blocks for more complex deceptive patches/combinations of deceptive principles within a deceptive patch. We also provide prior work/research or point to specific chapters within this dissertation for each deceptively influenced component of deception.

3.2.1 Deceptive Software Architecture

Deceptive software architecture applies deceptive techniques to the software architecture, or structural attributes, of a patch. This includes elements such as coding

Table 3.1.: Modeling the Space of Deceptive Patching. Gray Cell Background Indicates an Infeasible Category. Green Cell Background Indicates a Category that is Discussed in this Dissertation.

	Software Architecture	**System Input and Output**	**System Architecture**	**Deploy and Install Chain**
Mask	hide patch architectural components	hide patch input and output	hide the location of a patch	hide patch notifications
Repackage	hide the real patch within something else that is functional	active response, hide the behavior of a patch within other behavior	hide the location of a patch within something else	hide the notification of a patch within something else
Dazzle	make the structure of the patch confusing, obfuscate the code	random or confusing response	make the location confusing or random	make the notification confusing, puzzling, hard to read
Mimic	fake patch that exhibits structural characteristics of a real patch	fake patch that exhibits behavioral characteristics of a real patch	fake patch that shares location characteristics to real patches	fake notification that exhibits characteristics or a real patch notification
Invent	fake patch that appears real but it is completely made up	fake patch behavior that is completely made up	fake location that is completely fabricated	fake notification of a patch that is completely made up
Decoy	fake patch that is structured such that it will attract attention	fake behavior meant to attract attention from an adversary	fake location of a patch in a commonly visited area such that the patch is inviting	fake notification meant to attract attention

language, lines of code, number of basic blocks, variable names, memory size, basic block ordering, control flow and other elements that can be collected using static analysis tools or observation. It can also be said that these elements are gathered without executing the patch.

Mask Masking software architecture can be achieved by completely hiding one or more elements of a patch's structure. Examples include encrypting a patch, hiding the size and preventing the size from being calculated, or hiding the order of basic block execution [74]. Hiding these elements does not mean that they have to truly be invisible. For a patch to actually be applied, there is necessarily some change that the system must undergo. The key for hiding this information is to make it such that the software architecture elements cannot be detected by an adversary. Even with this relaxed definition, masking software architecture is infeasible given current technologies and standards of practice. The structural information about a patch can be leaked using side channel information. As a concrete example, if a patch's code is encrypted, it must be decrypted to be read on a machine for execution. Thus, an attacker can collect information on the commands being executed within an encrypted block of code by observing the instructions being called once this block is entered by a process.

Repackage Repackaging software architecture components can be achieved by enveloping these elements within another container. Examples include interweaving a patch within another program. Prior work in software diversity can be applied to repackaging software architecture [65, 67, 84].

Dazzle Dazzling software architecture components attempts to confuse an adversary. Examples include obfuscating source code [63, 64, 85–88]. Chapter 5 applies *dazzling* to software security patch architecture.

Mimic Mimicking software architecture components creates fake copies of real patches. These fake copies look and behave similarly to their real counterparts. Chapter 4 and work by Colbert et al. [63] and Crane et al. [81] explores mimicking software architecture in more detail.

Invent Inventing software architecture components applies new and fabricated behaviors, characteristics and concepts to the software architecture of patches. These elements should appear real. Collberg et al. research the impact of adding bogus code to programs in an effort to obfuscate [89].

Decoy Decoying software architecture components is similar to mimicking, but it is meant to attract attention away from the real elements. Decoys do not completely appear exactly as their real counterparts, but they have similarities such that they appear real. Chapter 4 applies decoy techniques to software security patch architecture.

3.2.2 Deceptive System Input and Output

Deceptive system input and output applies deceptive techniques to the input and output of a patch. This component of a patch represents the behavior of a patch. This can also be thought of as the stimulants to activate a patch, the program or machine state during patch execution and after patch execution. This is all the information that can be observed or calculated from a patch executing with inputs.

Mask Masking system input and output applies deceptive techniques to prevent input or output from being detected or measured. This suggests that before a patch executes and after a patch executes, program and/or machine state remain the same and changes cannot be measured or detected.

This deceptive principle is infeasible under system input and output as machine state must be altered (i.e. at the least the instruction pointer is incremented with a NOP instruction) once a line of code is run.

Repackage Repackaging system input and output involves enveloping any patch behavior within another vehicle such that the legitimate responses or input are not observed or detected. An example of a repackaging system input and output can be found in the REDHERRING tool [77, 78] as well as work by Crane et al. [81].

Dazzle Dazzling system input and output creates confusing responses or makes the response from or input to a patch confusing. One way to implement such an approach would be to provide random responses to input. Work by Stewart [90], Goh [91] and Balepin et al. [92] discuss responding to intrusions using various techniques, including dazzling.

Mimic Mimicking system input and output entails copying legitimate input or output, setting system state and using that as the response or input into a patch where the patch's state or response is different. Thus, the patch acts like and appears to an observer as another patch. Arujo et al. [77] and Crane et al. [81] apply this principle to deceive potential adversaries.

Invent Inventing system input and output creates elements to present a new reality. This principle provides the most flexibility to create new content and influence adversary decision making. Collberg et al. explain concepts and provide examples of bogus control flow in programs [63].

Decoy Decoying system input and output copies characteristics of legitimate system input and output to attract attention. Arujo et al. apply this principle to patch responses from honeypots [77].

3.2.3 Deceptive System Architecture

Deceptive system architecture applies deceptive techniques to the system architecture through a patch. This component of a patch represents the location of a patch within the system architecture. It can be said that system architecture is crafting, identifying and implementing where in the system a patch will be located. Studying where a patch can be implemented and identifying different locations where a patch can be implemented in turn can provide information about a patch as well as about the state of a system. Adding deceptive techniques to a patch's location could make the patch itself more difficult to exploit.

Mask Masking the system architecture of a deceptive patch involves concealing the exact location of a patch. This makes the patch location non-observable. This also has similarities to masking the software architecture of a patch. Because a patch changes software by adding, removing or editing some code, the location of a patch is infeasible to mask.

Repackage Repackaging the system architecture of a deceptive patch places a patch in another location where the new location serves a different purpose. Repackaging system architecture is also similar to repacking software architecture. Crane et al.

apply repackaging to create beaconing Return-Oriented Programming (ROP) gadgets [81].

Dazzle Dazzling the system architecture of a deceptive patch attempts to confuse an adversary regarding the location of a patch. This is a prime example of applying moving target defense techniques to patches. Making the location of a patch confusing and unstable makes exploiting the patch more difficult as the system may not consistently respond. MTD tactics and procedures, which have mainly been applied to computer networking, fall within this principle of deception [66, 93–97].

Mimic Mimicking the system architecture of a deceptive patch copies the location of a patch and implements or applies that in another location or system but shows some false components. One aspect of making a system seem to have a patch at a specific location, when in reality, all that is implemented is a shell. Mimicking system architecture is similar to mimicking the software architecture of a patch.

Invent Inventing system architecture of a deceptive patch involves creating a new reality about the location of a patch. This means that fake information about a patch is provided. This could mean that the patch itself is fake, similar to inventing software architecture, or that the location of a real patch is fake.

Decoy Decoying the system architecture of a deceptive patch involves placing false patches in locations that are attractive to an adversary. This idea is meant to shift attention toward these locations and away from other legitimate or more vulnerable areas. This type of patch is also closely related to decoy software architecture elements. Crane et al. apply this principle by implementing decoy ROP gadgets where the real gadget(s) are expected with beaconing capabilities and their security implications [81]. Chapter 5 addresses the application of *repackaging* and *dazzling* to system architecture.

3.2.4 Deceptive Deploy and Install Chain

Deceptive deploy and install chain applies deceptive techniques to the deploy chain of a patch as well as the installation process of a patch. This includes notifications before, during and after patch installation, the results of a patch, input into a patch and register values as a result of the patch execution. This component of deceptive patches is influenced by work studying deception within consumer advertising [98,99].

Mask Masking deploy and install chain elements of a deceptive patch involves hiding or concealing the notification information such as text, images, and sounds. This information notifies end users that a patch is available to install and provides status updates during as well as after the installation of a patch. Hiding this information and side channel leaks can be accomplished by not releasing any information about a patch, its effects on a system or the success or failure of a patch. Prior work has discussed the economic implications of hiding vulnerability disclosure and patch notifications [100].

Repackage Repackaging deploy and install chain elements of a deceptive patch will hide notifications about the presence of, installation of and success of a patch within other objects, code, data, etc. A simple example is to use stenography to hide a textual message about the contents of a patch within an image or within a separate patch's description. Chapter 5 discusses the application of this principle by repackaging old deploy and install chain notification in re-released patches.

Dazzle Dazzling deploy and install chain elements of a deceptive patch makes these notifications confusing to identify, view or understand. The real notification data could be written in a different language or provided to an end user in some way that takes time and resources to observe clearly/in a traditional manner. A simple example is to mix the letters in the notification text to make it unreadable without expending additional resources.

Mimic Mimicking deploy and install chain elements of a deceptive patch copies the syntax and semantics of other deploy and install chain instances from other patches. Using the same structure, wording and flow of information to the end user as another patch as well as fabricating this information is an example.

Invent Inventing deploy and install chain elements of a deceptive patch creates new realities about a patch being available or about the installation process. Fake notifications can be provided that create a new reality that vulnerabilities in code are being fixed by a patch. An example is to release a notification that says a patch is available for a vulnerability when there is no patch, or notifications during the installation of a patch can all be false.

Decoy Decoying deploy and install chain elements of a deceptive patch introduces fake notifications that are attractive to adversaries. These notifications appear promising in terms of being useful to accomplish an adversary's goal(s) and elicit further investigation.

3.2.5 Combining Deceptive Patch Elements

Each of the above deceptive patch categories, apart from those that have been identified as infeasible, can be combined with other deceptive patch categories. The combinations used to develop a deceptive patch are based on the needs of as well as the attack vector that has been identified by the developer. For example, if the patch's code is accessible by an attacker, the software architecture, system input and output, system architecture, as well as deploy and install chain elements are observable. Thus, deception could be applied to one or multiple patch elements to deceive attackers. If the patch's code is not accessible, then deception could only need to be applied to the system input and output to deceive attackers. The order that multiple deceptive techniques are applied to a program is dependant on the needs of the developer and the program being patched. For example, if deception is being applied to both the system input and output and the software architecture, and the attacker can view the patch, applying deception to the system input and output first and then applying deception to software architecture to hide the real patch and/or show false elements layers the deceptive techniques that have been applied.

Prior work combines multiple categories of deceptive patches. REDHERRING combines both dazzling and mimicking system input output [77]. Faux patches, explained in more detail in Chapter 4, combines mimicking and decoying software architecture.

3.3 Deceptive Patching Approach Discussion

As part of this research, key details that influence how deceptive patches are designed and implemented must be discussed. This section compares perspectives of patching code and explains why we believe our approach best accomplishes the goal of influencing an attacker's decision making.

3.3.1 Deceptive Patching vs. Traditional Patching

The main goal of traditional patches is to remove the vulnerability from the code. First, developers detect or are notified of a vulnerability or existing exploit for their program. If they have access to the source code, they can make the necessary changes to address the vulnerability, making any exploit(s) against that vulnerability harmless. If they do not have access to the source code, an exploit signature can be created and applied to a firewall to detect high level elements of the exploit as it travels on the network. Traditional patches are beneficial because they improve the security of a function by addressing the vulnerability when implemented correctly and preserve the main functionality of the code. Simultaneously, traditional patches can *weaken* systems because they *leak information to an attacker about the system's state.* These patches expose flaws to attackers that they can utilize to gain elevated privileges, steal data and/or perform malicious unauthorized actions [42].

Deceptive patches have two primary goals. The first is to address the vulnerability present in the code. This goal has the same security benefits as traditional patches. The second goal is to influence an attacker's decision making. Deceptive patches can themselves be fake or complex. These types of patches can influence an attacker to develop exploits for fake vulnerabilities, for an incorrectly assumed vulnerability, or waste time and resources. Deceptive patches can also return data that attackers or malicious programs expect or believe to be confidential, is fake, or is misleading based on their complexity. Because the nature of these patches is to fix the issue as well as deceive an attacker trying to exploit the vulnerability, they may not expose the vulnerability to an attacker as easily as traditional patches.

Keeping an attacker's interest is important for the success of deceptive patches. These patches also have a psychological effect on attackers. Future attacks could be prevented or attackers may approach systems much more cautiously if they have knowledge that deceptive patches have been implemented in the system they are attacking. Without sure knowledge of how the deception works or a way to verify the information they receive, malicious actors will be more wary to attack systems.

Thus because of the added benefits of deceptive patches, namely the psychological effect and the potential for counter-intelligence gathering by defenders, deceptive patching has the potential to improve program security more than traditional patching. Thus, this research will analyze deception's application to patching and potential impact on software security.

Deceptive Patching Limitations Deceptive patches are not without limitations. A list of deceptive patch limitations follows:

1. The potential increase in time to develop deceptive patches. Because deceptive techniques must be studied and analyzed for different types of vulnerabilities, creating a deceptive patch may be more involved compared to traditional patches. Researching and developing ways to optimize deceptive patch creation will reveal techniques to decrease the development time.

2. The risk of counter-attacks. An attacker with knowledge that deception exists on a system can purposefully use exploits that they know a defender expects and give defenders the false idea their defenses are effective.

3. The lack of concealing the general location of actual vulnerabilities. Deceptive patches can dazzle the legitimate patch locations by injecting many false patches, potentially increasing the workload required to identify actual vulnerabilities based on a patch. This does not completely hide the location of the actual vulnerability.

4. The increase in patch size. Adding fake code to legitimate patches will increase the memory size of a patch.

3.4 Cyber Kill Chain Analysis

In this section, we map each component of a patch onto the cyber kill chain model and show where deceptive techniques and tools that are associated with a particular component impact the kill chain [7]. The cyber kill chain model captures the series of steps an attacker performs to collect target information, develop and release an exploit and maintain presence in the compromised system. Table 3.2 provides a general overview of major types of patches in prior work as well as those that are addressed in this dissertation (listed by chapter title or section heading). Listing a deceptive patch concept or technique at a specific stage in the cyber kill chain suggests that the patch impacts decisions made during this phase.

Many of the deceptive patch techniques and concepts impact the *reconnaissance* phase of the cyber kill chain. This occurs because exploits are developed based on a patch. When deception is applied, attackers will be influenced during the information gathering phase of the exploit development kill chain. Attackers use a patch to develop their exploit, so as they are studying the patch and attempting to understand its behavior, components, and information gathering, deception will influence the information they gather and their subsequent actions. This analysis is important because interrupting the cyber kill chain early affects the future steps in the sequence. Because deception impacts components within a patch used by the attacker to make decisions, and deception is observed by an attacker, the information gathered by an attacker may be deceptive.

One observation from this table is that the *weaponization* phase is not affected by deceptive patching. The attacker is not prevented from creating an exploit and in some instances, depending on the environment of the patch may be baited to make an exploit. This suggests that deceptive patching should be considered a supplemental defensive mechanism that is implemented alongside more traditional defensive techniques to provide resiliency against attacks.

We also note that deceptive patches impact the Observe, Orient, Decide and Act (OODA) loop [102] decision making model in a similar manner compared to the cyber kill chain. The steps within this model are observe, orient, decide and act. This model describes the decision making process of actors engaged in conflict with the premise that completing the loop more quickly and accurately than an adversary results in a successful action taking place and gaining momentum. Deceptive patches impact the observe and orient stages of the OODA loop, influencing the remaining two steps in the decision making process. The consistency of our deceptive patch model with both the cyber kill chain and the OODA loop suggest that the model accurately represents the deceptive patch space.

3.5 Modeling the Intended Effect of Deceptive Patching

Deceptive patching can also be modeled in terms of the goals and outcomes. In this section we explore how each element of a patch, when deceptively implemented, impacts an attackers time to discover a vulnerability based on a patch as well as their time to develop an exploit. We first explore elements that make up an attackers' timeline to attack and analyze how elements of a deceptive patch impact this timeline.

Table 3.2.: Mapping Deceptive Patching Tools and Techniques onto the Cyber Kill Chain. Green Background Indicates Tool or Technique Discussed in this Dissertation.

	Software Architecture	**System Input and Output**	**System Architecture**	**Deploy and Install Chain**
Reconnaissance	Ghost Patches [8]	Active Response [76, 91, 92, 101], HoneyPatches [77]	MTD Patching	Deceptive Dispatcher
Weaponization				
Delivery	Data Patches [12, 34]			
Exploitation	Legitimate Patches	Active Response, HoneyPatches	MTD Patching	Deceptive Dispatcher
Installation			MTD Patching	
Command and Control		Active Response, HoneyPatches		
Actions on Objectives		Active Response		

3.5.1 Attacker Timeline to Exploit

An attacker follows a generic timeline as s/he develops an exploit based on a patch. Each component of the timeline builds on the prior segment such that each element's time is the cumulative effect of all prior elements.

Time to Identify Patch The time to identify a patch encompasses the time to identify that a patch for some software is available. Traditionally, this time is minimal as when a patch is publicly available, a notification is also released. This time applies to both benign end users as well as malicious adversaries.

Time Lag This is the time window between when a patch is released and when a benign user actually downloads, installs and applies the patch. Bambenek explains that during this window, attackers use the patch to develop and release exploits, taking advantage of the unpatched systems [103]. Forced updates and hot patches attempt to shorten this time frame, making exploit generation more difficult.

Time to Reverse Engineer The time to discover the vulnerability that a patch is fixing is the segment of time that begins once a patch is discovered and ends when the vulnerability being fixed is identified. This time period is unique to patch-based exploit generation as the patch itself fixes the vulnerability but also identifies the vulnerability being fixed. This is also the time segment that is immediately impacted by deceptive patching techniques. Lengthening the time to reverse engineer a patch to identify the legitimate vulnerability being fixed provides more time for end users to update their system.

Time to Create/Generate an Exploit The time to develop an exploit is the time segment that begins once the vulnerability is discovered and ends once a reliable exploit is developed. A reliable exploit is one that consistently exploits the vulnerability being fixed. Once an exploit is developed, this time segment stops.

Time to Install Patch This is the time for benign users to install a patch on their system. This time includes the time to edit, add or remove files, perform checks, start and restart the system and to show that the patch was successfully installed on the machine. This period also includes the time for end users to verify the compatibility of patches with other programs running on their machines, legacy programs and standards [104].

Time Point Patch Release Identified The time point a patch release is identified is the exact time that either a benign user or malicious adversary discovers that a software security patch is available for download. This time point can be different for each class of user.

Time Point Patch Executable Downloaded The time point a patch executable is downloaded represents the exact time that a benign user downloads a patch to install it on their system. We only represent the benign user's time because we assume an attacker downloads the available patch quickly after discovery. Automatic updates using an update center that pulls patches from a central server to each individual machine attempts to decrease the time from patch release to patch download.

Time Point Vulnerability Identified We identify the time point in which the vulnerability being updated by the patch is identified. This occurs once the patch is successfully reverse engineered. This is an implication that the code has been statically and/or dynamically analyzed.

Table 3.3.: Deceptive Patch Timeline Symbols

Time Symbol	Description
T_P	Time to identify a patch has been released
T_L	Time between when a patch has been released and when it is installed by benign end user
T_I	Time to install a patch, including the time to verify the patch is compatible with software on the machine to be updated
T_{RE}	Time to reverse engineer a patch
T_{CE}	Time to create/generate and exploit
T_A	Time to attack
T_{PRI}	Time point at which the patch release was identified
T_{PED}	Time point at which the patch executable was downloaded
T_{VI}	Time point at which the vulnerability was identified
T_{ED}	Time point at which exploit was developed

Time Point Exploit Developed The exact time when an exploit has been implemented and tested and reliably exploits the vulnerability being patched.

3.6 Chapter Summary

This chapter presents and discusses a model of what composes a software security patch, applies deceptive principles to this model and then maps specific deceptive patch tools and concepts onto the cyber kill chain. We discuss where gaps are present in the field and how they are addressed by concepts, implementation and analysis presented in this dissertation. We also analyze an economic model of deceptive patches and visualize how they influence patch-based exploit generation.

4. SOFTWARE ARCHITECTURE

This chapter presents an approach to adding simulated patches to code. Using a compiler-based tool, fake patches are inserted that mimic integer input validation vulnerability patches. The choice to analyze deceptive patches for input validation vulnerabilities is based on the common patching structure used to patch legitimate input validation vulnerabilities in software [38]. These patches suggest that a vulnerability is present at a location where the vulnerability is not. We also analyze program runtime impact to identify how fake patches affect legitimate program execution as well as dynamic analysis runtime to measure the effect of fake patches on software path enumeration, which can be used to develop exploits. Finally, we discuss limitations to our approach. [1]

4.1 Motivation

As described in Chapters 1 and 2, attackers use techniques such as binary diffing and control flow analysis to reverse engineer patches and develop exploits for the vulnerability being fixed. As analysis techniques improve, the speed of patch-based exploit development improves, giving end users less time to protect their systems by applying the patch. One method to create more time for end users to protect their systems is to use deception to influence attackers, causing them to expend more time analyzing the patch compared to the time to analyze currently implemented patches. We implement a proof-of-concept compiler-based tool to insert deceptive patches that *mimic* legitimate patches.

[1]Portions of this chapter are taken from Ghost Patches: Faux Patches for Faux Vulnerabilities [8]

4.2 Technical and Approach Background

Deception Fake patches are an application of showing the false by *mimicking* and *decoying* and hiding the real by *dazzling*. They show the false by including characteristics of real patches, mimicking a real patch and attracting attention away from traditional patches as a decoy. Fake patches hide the real by reducing the certainty of which patches are real and which are decoys. This added uncertainty adds a layer of protection to legitimate patches by causing a greater potential for increased workload to exploit vulnerabilities based on patches.

LLVM Lower Level Virtual Machine, LLVM, is a "collection of modular and reusable compiler tool chain technologies [105]." This tool started as a research project at the University of Illinois at Urbana-Champaign in 2000 by Vikram Adve and Chris Lattner. The ghost patch implementation uses an LLVM pass to insert faux patching. We use LLVM because of its versatile front end compiler options, removing restrictions on source code language to implement and apply faux patches.

Symbolic Execution Automatic exploit techniques use symbolic execution to generate malicious inputs to programs [38,44]. Symbolic execution uses branch statement conditional expressions to generate paths throughout a program. By generating sample input values for each branch of a conditional statement, each path in a program can be covered without the computational strain of a brute force approach. Once symbolic execution is completed, all input and variable values that form a path through the program are known. Comparing the signatures from an unpatched program and a patched program can identify changes in the branches within each program and attackers can use these discrepancies to develop an exploit. Symbolic execution is applied to dynamic analysis within KLEE [106]. This tool creates symbolic execution signatures. We use the runtime of KLEE to indicate the amount of work necessary to develop an exploit. An increase in runtime suggests that more patches throughout a program were discovered by the tool. This increase in paths to enumerate because of faux patches ideally results in a longer analysis time, which correlates to an increased workload to identify the legitimate path and associated patch.

Input Validation Vulnerabilities This work targets input validation vulnerabilities. A common patch to these types of vulnerabilities is to add boundary checks in the form of if-statements [38]. Thus, given a patched and unpatched program, a diff between the two programs will show additional branch statements in the patched version. These branch statements can be used to then determine input values that will exploit an unpatched program.

4.3 Ghost Patching Approach

This research studies how a fake patch can be implemented in conjunction with a traditional patch and measures its impact on program analysis and runtime. These fake patches should alter the control flow of a program, but not the data flow of information. Thus, given two programs, one with a ghost patch and the other with a traditional patch, the final output should be identical.

Our approach is based on a common patching behavior that input validation vulnerabilities are fixed by adding conditional statements that validate the value of variables that can be tainted by malicious input [38]. Thus, to deceive attackers, we add fake patches to code that mimic these input validation conditional statements, making exploit generation using patches more resource intensive.

4.3.1 Threat Model

We consider attackers who are using patches to develop exploits and have access to both patched and unpatched versions of a program, and can control and monitor the execution of both as our threat model.

Ghost patching is designed for input validation vulnerabilities that have not been discovered by the public or do not have a widely available exploit. If there are scripts that already exploit a well known vulnerability, ghost patches can still be applied but with less effectiveness. Public exploit databases[2] or "underground" forums could be monitored to determine if exploits have been developed.

We specifically look at input validation vulnerabilities that involve integers. These vulnerabilities can be exploited because of a lack of boundary checking and can cause subtle program misbehavior through integer overflows or underflows.

Finally, ghost patches target input validation vulnerabilities in enterprise scale systems. Real time systems are not suitable for ghost patches because adding control flow statements could increase the runtime of code, potentially violating time constraints of functions in these systems.

4.3.2 Properties of Ghost Patches

This work applies concepts from decoy documents to deceptive patches. Decoy documents are fake documents inserted into a file system or on a personal computer and are meant to intentionally mislead attackers. These documents also *mimic* real documents and are *decoys* meant to attract attention away from critical data. Bowen et al. and Stolfo et al. have conducted research on decoy documents [50, 70] and created a list of properties that decoy documents should embody. We slightly modify

[2]E.g. https://www.exploit-db.com/

Table 4.1.: Fake Patch Properties

Property	Explanation	Implementation Effort
Non-interfering	Fake patches should not interfere with program output nor inhibit performance beyond some threshold determined on a case to case basis.	Experimentation
Conspicuous	Fake patches should be “easy” to locate by potential attackers.	Easy
Believable	Fake patches should be plausible and not immediately detected as deceptive.	Easy
Differentiable	Traditional and fake patches should be distinguishable by developers.	Experimentation
Variability	Fake patches should incorporate some aspect of randomness when implemented.	Easy
Enticing	Fake patches should be attractive to potential attackers such that they are not automatically discarded.	Experimentation
Shelf-life	Fake patches should have a period of time before they are discovered.	Experimentation

these properties and present in Table 4.1 our list of fake patch properties as well as whether the property is trivial to implement or requires further experimentation.

4.3.3 Types of Faux Patches

Faux patches could be generated using a variety of different structures. Faux patches could be generated using a conditional statement with code in the body of the statement. We explore this construction in more detail as we discuss our implementation of faux patches. Faux patches can also be constructed as a loop.

These patches would perform some calculation that does not impact the data flow throughout a program but would alter the control flow. Because loops iterate multiple times during a program run, their impact on program runtime can be significant. Further analysis could identify acceptable loop conditional operators and values that impact program runtime within an acceptable threshold. Faux patches could also be one or more lines of code that calculate a value within a program. One example is to apply the concept of opaque predicates to create expressions that are trivial to calculate but difficult to understand [63]. Finally, faux patches could be implemented as entire functions. This type of faux patch inserts fake functions calls that appear to perform operations that do not impact data flow.

4.3.4 Implementation Properties

The implementation of fake patches applies deception to patching because it attracts attention away from a traditional patch, but does not impact the data flow of the function being patched. Fake patches should be designed such that they are not marked as *dead-code* and removed from the binary as a result of compiler optimization nor should they be trivial to identify by attackers. These patches should also address the properties outlined in Table 4.1. Implementation components of a fake patch should at a minimum include at least one randomly generated value and a conditional statement. Other implementation specifics depend on the actual program being patched.

Control Flow Fake patches having conditional statements that alter control flow will make them apparent to attackers using static and dynamic analysis tools. This addresses the *conspicuous* property. This also mimics the trend of patches for input validation vulnerabilities.

Mimicking this trend could deceive attackers by showing changes that are expected but fake, addressing the *enticing* property. Experimentation will show how fake patches affect overall program runtime, addressing the *non-interfering* property. We implement fake patch conditional statements such that they include the destination or left-hand-side of an LLVM intermediate representation *store* instruction in the original program mathematically compared to a randomly generated value. The use of a random value addresses the *variability* property.

We form the body of if-statements by adding code that solves different mathematical expressions with the original program's value as input. These expressions do not alter the value of the legitimate variable. Thus, data flow is preserved. The body of fake patch statements should be plausible for the program being patched. This suggests that the body of a fake patch should be developed based on the behavior of the program being patched.

4.3.5 Post Testing

After applying a ghost patch to software, further testing should be conducted for the following:

1. Evaluating ghost patch impact on software runtime and program memory (i.e. lines of code).

2. Verifying ghost patch does not introduce incompatibilities by applying unit testing.

A ghost patch should be evaluated for its impact on the program's performance to determine if it is feasible. This determination is dependent upon each program and the execution environment of the program. The memory impact of a ghost patch should also be considered. The size of a ghost patch should be reasonable for end users to download and apply to vulnerable systems. Developers should establish an upper threshold such that the feasibility is measurable and can be validated. Conjectures about patch size and acceptable runtime are outside of the scope of this research. We do analyze the statistical impact of ghost patches on program runtime and program analysis.

4.3.6 LLVM Workflow

The workflow of our LLVM prototype begins with a traditionally patched file (we assume developers have previously created a traditional patch). First, this traditionally patched file is compiled using *clang* [107]. This creates intermediate representation bytecode of the traditionally patched program. Next, this file is compiled a second time, applying our ghost patch LLVM pass. This pass adds one or more fake patches, which are also implemented in bytecode, after store instructions in the traditionally patched program's bytecode.

We choose store instructions because they are a natural instruction that propagates data throughout a program. The store instruction transfers data from one location to another. Thus, placing fake patches after store instructions that identify boundaries directly after the data is stored in a variable provides the most spatial and temporal locality. By this we mean placing the patch right after the store instruction provides temporal locality in that the time between a vulnerable segment of code executing and the patch executing is minimized. By spatial locality, we mean that there is minimal space between the location of the vulnerable segment of code and the location of the patch. Other options for loading points of faux patches include load instructions, because this is another location where data is propagated through code. Also, load instructions could propagate malicious code, these could be insertion points. A final general option is specific patterns of instructions. An example is a load followed by an add followed by a multiplication. We use conditional statements because these statements create branches within code that can be easily detected.

This stage creates a new ghost patched program. Next, this ghost patched program is compiled into binary using the *clang* compiler. If the file being patched is part of a larger project, the build tool for the project should be mapped to clang to ensure the project gets compiled with the correct flag(s). After the ghost patched code is compiled, the patched and unpatched (this file is before any traditional patch has been applied) binaries are supplied to a binary diff tool, such as *bsdiff*, to create a patch file that can be distributed and applied to unpatched programs. A work flow diagram of this process is shown in Figure 4.1.

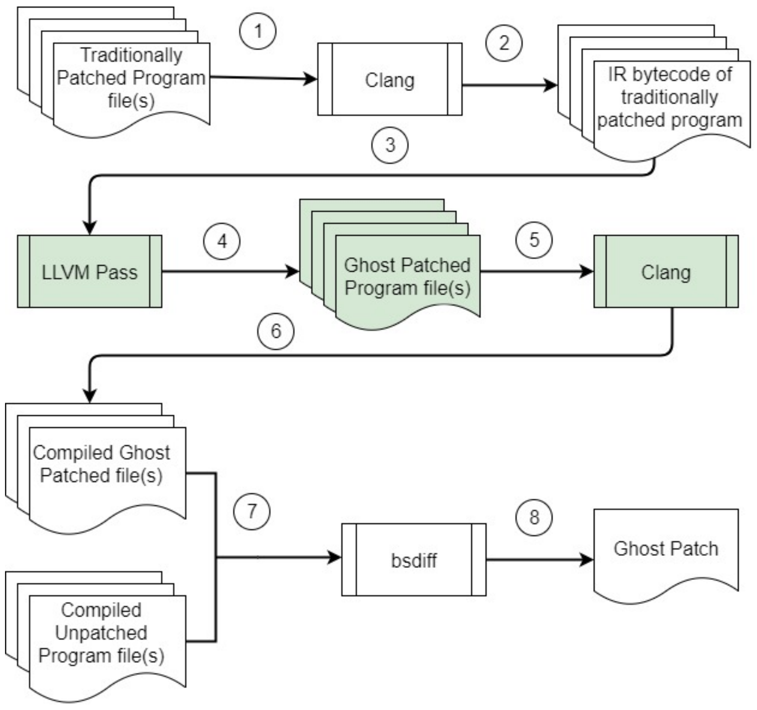
Traditionally Patched Program file(s)
1
Clang
2
IR bytecode of traditionally patched program
3
LLVM Pass
4
Ghost Patched Program file(s)
5
Clang
6
Compiled Ghost Patched file(s)
Compiled Unpatched Program file(s)
7
bsdiff
8
Ghost Patch

Fig. 4.1.: Complete Flow to Create a Ghost Patch Using LLVM and *bsdiff*. Green Shading Indicates Steps Added to Software Patching Process.

4.3.7 Implementation and Testing

We implemented a proof-of-concept that addresses input validation vulnerabilities involving integer variables. We believe our approach can be extended to other variable types and data structures without loss of generality. Our implementation uses LLVM and is about 900 lines of C++ code.

4.4 Ghost Patch Evaluation

The prototype of the faux patch program was developed using an LLVM pass on an Ubuntu 14.04 x86_64 virtual machine with 2 cores and 4GB RAM. We used LLVM (version 3.4) to develop our pass because it includes a front end compiler supports optimizations to be developed and applied to programs agnostic of the coding language.

Fake patches increase the number of branches in a program because of the conditional statements that are added at *store* instructions. One method to quantify the impact of fake patches on both program runtime and program analysis is to use symbolic execution. Symbolic execution identifies paths through a program using symbolic inputs based on conditional statements throughout a program [38]. Attackers can use symbolic execution to identify new paths through a program as well as the input values that cause these paths to be traversed. This analysis can be used to automatically identify inputs that satisfy the conditions to execute the code of a patch. This would provide attackers with the necessary information to develop an attack that exploits the vulnerability being fixed in unpatched programs by using input values that satisfy patched code conditions in unpatched programs. We apply symbolic execution to deceptive patches and use both the number of paths enumerated as well as the analysis runtime to provide insight into the required workload to analyze faux patched and unpatched programs. Specifically, we use KLEE [106] (version 1.3) to perform this analysis because of its compatibility with LLVM.

4.4.1 Simple Example

We evaluated our approach using the example below, which allows a user to enter two values and then copies each value into an integer variable and lacks input validation code. Then, some operations are performed and the results returned.

```
int calculate(int alpha, int beta);


int main(){
        int a,b,c;
        int d = 9;

        printf("Enter a value: \n");
        scanf("%d", &a);
        printf("Enter another value: \n");
        scanf("%d", &b);

        c = calculate(a,b);
        printf("Value of C: %d\n",c);

        a = b + d;
        if(a > 27)
                c = c * d;
        else
                b = a - b;
        d += d;
        return a;
}

int calculate (int alpha, int beta){
        if(alpha > 88)
                return (alpha + beta);
        else
                return (alpha * beta);
}
```

We also showed the results of our approach using approximately 15 examples from a publicly available benchmark for the KLEE symbolic execution tool. This benchmark was created by NEC Laboratories America with the purpose of research and testing. [3] This suite includes programs with single source files that use input to perform various options such as filling in values of an array or calculating the sum of a variable in a loop.

Experimentation To evaluate our approach, we compare the length of time for KLEE [106], a symbolic execution, dynamic analysis tool, to analyze a legitimately patched and faux patched version of the code. We use the runtime of KLEE to suggest the impact of a faux patch on attacker workload for exploit generation. We take advantage of the fact that each new branch will be analyzed because fake patches are indistinguishable from traditional patches from a software perspective.

To show the effect of our approach on program analysis, we evaluate whether the time to dynamically analyze traditionally patched code is significantly different

[3]https://github.com/shiyu-dong/klee-benchmark

statistically when compared to dynamically analyzing fake patched code using a t test. We also evaluated program runtime using this same experimental structure to determine a fake patch's effect on program performance.

4.5 Results

4.5.1 Runtime Analysis

Using our simple code example, we collected runtime values using the *time* command for both the original program and a faux patched program. Figure 4.2 shows the difference in program runtime between a fake patched program and the unpatched program across 100 executions. Using this data, we determined the statistical significance of this difference in runtime using a t test. We concluded that there was no statistical significance between the runtimes for the original program and the faux patched program.

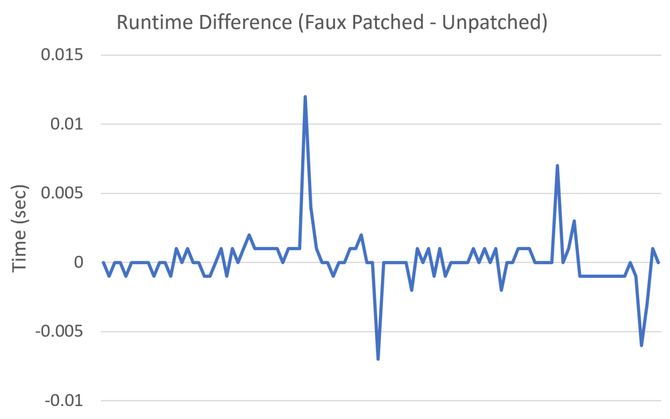

Fig. 4.2.: Difference in Faux Patched vs. Unpatched Simple Program Runtime

Table 4.2 provides the t-test values that provide insight to the statistical significance between the runtime of a faux patched and unpatched program. If the value in the t-stat column is greater than the value in the t-Critical two tail value column

Table 4.2.: t-test for Program Runtimes for Faux Patched vs. Unpatched Program (100 Runs per Program)

Program Name	t-stat value	t-Critical two tail value
ex12	2.114756796	1.973612462
ex17	0.140895756	1.972079034
ex21	-0.952802154	1.972017478
ex23	0.523293161	1.972204051
ex34	-0.123895226	1.972017478
ex42	-0.082101492	1.972017478

or less than the negation of the value in the t-Critical two tail value column, then the sets being measured have a statistically significant difference. For this analysis, this means that if one of the conditions holds true, then the program runtimes differ significantly. This would suggest that faux patches have a statistically significant impact on program runtime.

Based on this table, one program's runtime differs significantly between a faux and unpatched program. This statistically significant difference in runtime is caused by the program not having a significant number of instructions. Thus, adding faux patches almost doubles the program's size, increasing program runtime.

4.5.2 Program Analysis

We collected values for the runtime of KLEE using the *time* command as it analyzed an unpatched, traditionally patched and faux patched version of our simple code example. Figure 4.3 represents the runtime for our simple program across 100 executions for faux patched, unpatched and traditionally patched versions. A t test using these values revealed that there is a statistical significance in KLEE's runtime between a traditionally patched program and a faux patched program. This suggests that it is more resource intensive to analyze a faux patched program compared to a

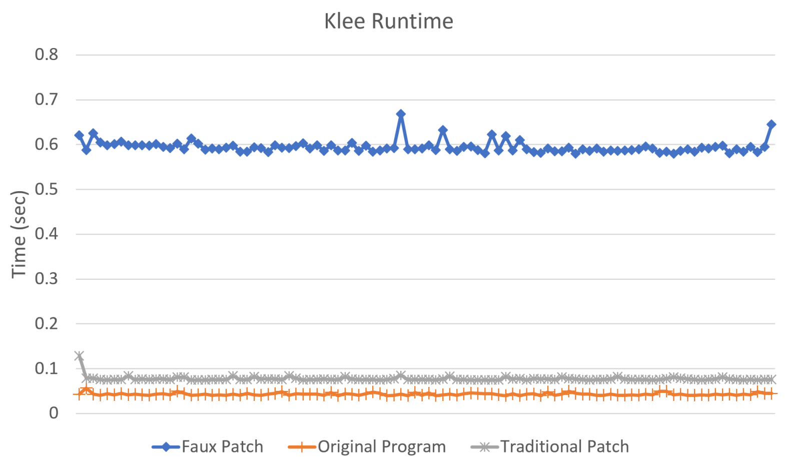

Fig. 4.3.: KLEE Runtime Analysis for Simple Program

traditionally patched program for our simple example, thus, analyzing ghost patches would also require more resources.

We also show the impact of faux patches on the runtime of an example symbolic execution tool, KLEE [106] for programs in the KLEE-benchmark suite. The runtime of KLEE is an indication of the impact faux patches have on program analysis. We also show the differences in the number of paths between an unpatched and faux patched program. We omit outliers from graphs for visualization purposes. We also only include programs where the difference in the number of KLEE paths between faux patched and unpatched programs is nonzero. Of the 30 programs, there are 16 programs that result in differences of KLEE paths between faux patched and unpatched programs. Of these 16 programs, 8 exhibit an increase in the average number of paths enumerated and 8 exhibit a decrease in the average number of paths enumerated. After further analysis, these programs that increase the average number of paths have symbolic variables that directly control the number of times a loop executes. Figure 4.5 shows the programs that exhibit a change in the number of KLEE paths excluding outliers. These outliers are programs that have a difference in the number of KLEE paths that is greater than 48. We remove these values from the graph but include all programs with differences in the number of KLEE paths enumerated in Table 4.3. Within this table, there are also negative values. These values represent programs with fewer paths enumerated by KLEE in their faux patched version compared to the unpatched version. This occurs because the faux patched version has optimization techniques applied based on how the compiler adds faux patches.

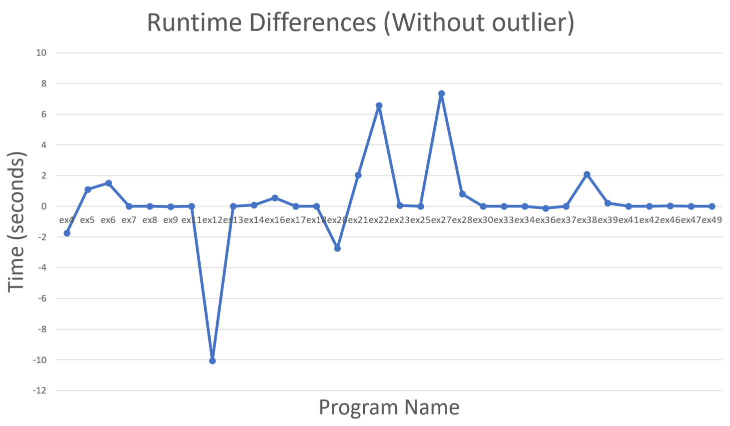

Fig. 4.4.: KLEE Runtime Analysis for KLEE-benchmark Programs in Seconds

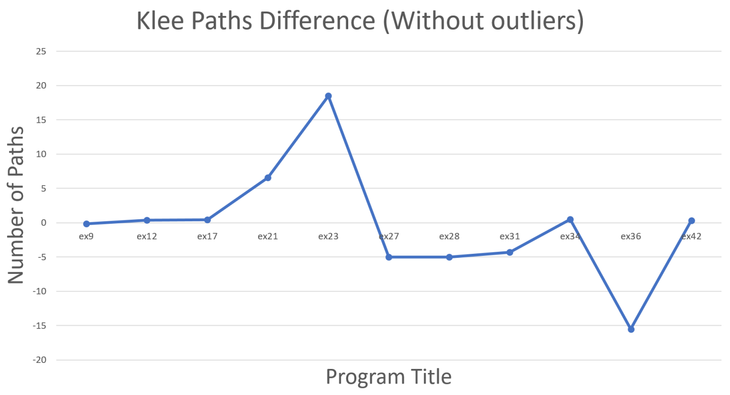

Fig. 4.5.: KLEE Path Analysis for KLEE-benchmark Programs

We also performed an experiment testing correlation between the number of faux patches added to a program and the impact on KLEE's runtime, number of paths

Table 4.3.: KLEE Path Differences for Programs in KLEE-benchmark Suite

Program Identifier	Average Number of Conditional Statements Added	Average Number of Paths Added from Faux Patch KLEE Analysis
ex9	6.4	-0.15
ex12	3	0.4
ex16	12	221.4
ex17	11.5	0.45
ex20	9	-48.65
ex21	4	6.6
ex23	7	18.5
ex25	187.65	797.15
ex27	9.25	-5
ex28	3	-5
ex31	4	-4.3
ex34	10	0.5
ex39	4	-582.85
ex42	3	0.3
ex49	6	-51.85

and number of tests generated. We use this test to provide insight into faux patches' impact on workload required to analyze a program.

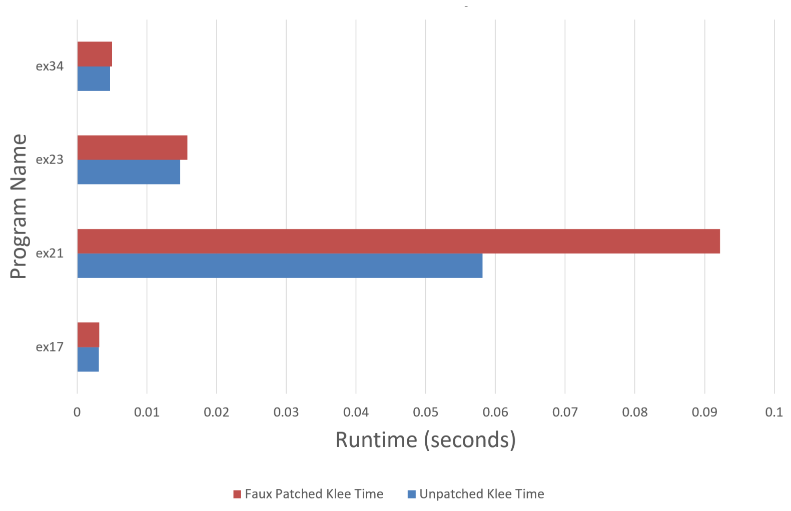

Fig. 4.6.: KLEE Runtime Analysis Increase

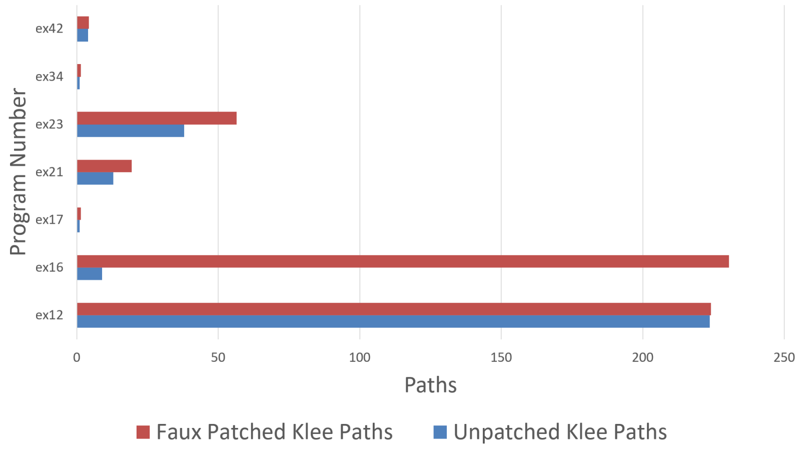

Fig. 4.7.: KLEE Path Enumeration Increase

Figures 4.8, 4.9, and 4.10 show the change in runtime, paths enumerated and tests generated respectively using KLEE when the number of conditional statements added increases for each *store* instruction. We compare the program analysis values when we

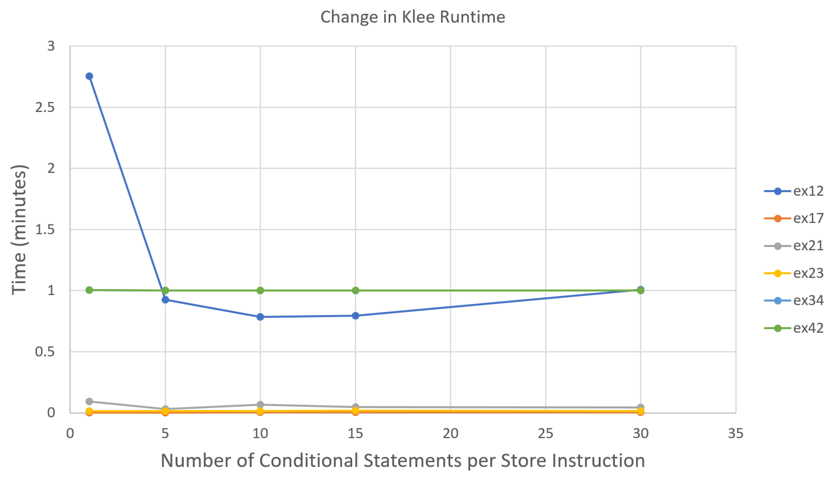

Fig. 4.8.: KLEE Runtime Analysis for KLEE-benchmark Programs Given Increasing Faux Patch Additions

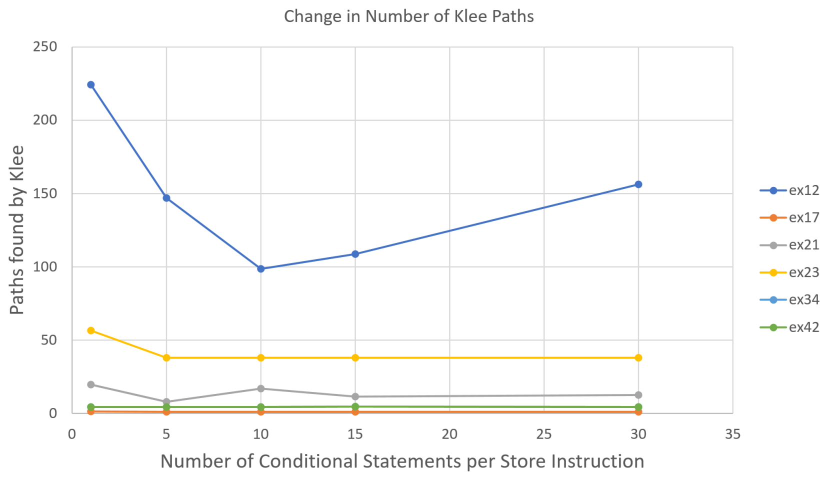

Fig. 4.9.: KLEE Runtime Analysis for KLEE-benchmark Programs Given Increasing Faux Patch Additions

add 1, 5, 10, 15, and 30 conditional statements per store instruction (this is equivalent to adding 2, 10, 20, 30, and 60 new branches or paths per store instruction). These graphs show that increasing the number of faux patches added to a program does not generally increase the analysis runtime, paths enumerated, nor tests generated. This suggests that the workload required to analyze faux patched programs is not

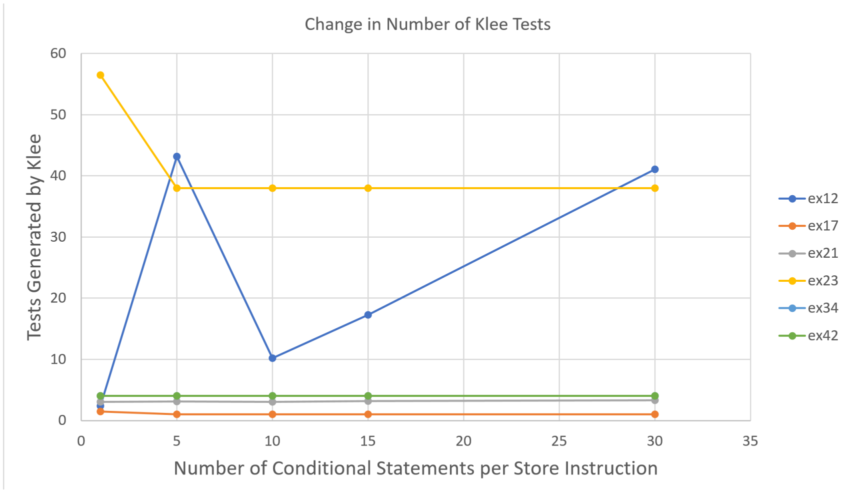

Fig. 4.10.: KLEE Test Analysis for KLEE-benchmark Programs Given Increasing Faux Patch Additions

positively correlated to the number of faux patches in a program. We also provide the t-test values for KLEE runtime values for faux patched and unpatched programs. Table 4.4 shows that there are two programs where the difference between KLEE's runtime for each program is statistically significant (ex23 and ex34). These variables in these programs that are symbolically represented are used in store instructions throughout the code or directly set the value of variables that are used in store instructions. Because faux patches use the store instruction to insert fake patches, the symbolically represented variables impact the path taken at a faux patch. Thus, symbolically executing the program with faux patches adds conditional statements that are directly impacted by the value of the symbolic variable.

4.6 Discussion

Based on testing of the KLEE-benchmarking suite, this section provides a more in-depth analysis of how ghost patching impacts these programs. Of the 30 programs tested, the average KLEE analysis runtime for analyzing faux patched programs in-

Table 4.4.: *t*-test for KLEE Runtimes for Faux Patched vs. Unpatched Program (20 Runs per Program)

Program Name	*t*-stat value	*t*-Critical two tail value
ex12	0.31772884	2.024394164
ex17	-0.814844341	2.063898562
ex21	1.299052449	2.028094001
ex23	3.856578304	2.085963447
ex34	-16.77526962	2.032244509
ex42	1.516182129	2.079613845

creased when compared to unpatched programs. After further analysis, this increase is attributed to the increase in instructions that are added by the faux patch protocol, not an increase in the number of paths to analyze. This is supported by our analysis which shows the number of paths KLEE enumerates for faux patched and unpatched programs is equivalent but the number of instructions KLEE executes increases for faux patched programs.

Programs that increase KLEE's runtime analysis when faux-patched have similarities that suggest key program characteristics for efficient faux patch application. Each of the programs with an increase in KLEE analysis runtime contain loops where the number of iterations is dependent, either directly or indirectly, on the symbolically modeled variable. In these programs, the symbolically modeled variable is the input value of the program. In the two programs where a larger increase in the average number of paths between faux patched and unpatched programs is observed, the input variable that is symbolically modeled by KLEE is directly responsible for the number of iterations that the loop performs. Thus, as the value used for the symbolically modeled variable changes during each program execution instance, so does the number of paths throughout the program. The third program's input variable indirectly impacts the number of loops that are executed, thus, the impact on the number of paths is reflected by a small increase in the number of KLEE paths comparing faux to unpatched versions.

The one consistent increase for all program is the positive increase in the number of branch statements in the code. At a minimum, this approach adds noise to code and when paired with legitimate patches, increasing the workload of attackers using binary diff and static analysis approaches.

We also identified the types of conditional statements that were inserted as faux patches. Based on the three programs that saw an increase in KLEE analysis runtime, there is no definitive ordering or ranking to the statements that are inserted into the code and an increase in analysis runtime or the number of paths. This suggests that the impact of faux patches is based on a combination of the selected value used in the conditional statements we well as the comparison operator used in these statements.

4.6.1 Achieving the Ideal

Based on our analysis, faux patches that are inserted at store instructions are best applied to programs with input variables that are written to use a store instruction. This corresponds to using the assignment operator, "=," instead of a library call such as *memcpy* or *strcpy*. An additional characteristic of an ideal program is one with one or more loops where the number of iterations is delineated by the value of an input variable. Another characteristic of ideal programs is the store instruction is in the body of a while loop. Thus, the faux patch is evaluated each time the loop executes, which increases the number of branch points in the program. We see this behavior in Figure 4.7 with program ex16. The increase in the number of paths is caused by the value of the input variable controlling the number of loop iterations as well as the input variable being written to in a store instruction within the body of the loop. This same behavior is also evident in the same figure with program ex23, which also resulted in an increase in the number of paths enumerated for faux patched programs compared to unpatched programs.

Patch Obfuscation There are limitations associated with ghost patches that could provide attackers an advantage in identifying fake patches and analyzing fake patched code. Attackers could use exploit generation tools that perform analysis in parallel [38] to distribute the analysis load across multiple machines and optimize exploit generation. One solution is to develop fake patches that increase the length of each path in a program such that tools are unable to identify the legitimate patch and as a result cannot develop an exploit. Another solution is to implement polymorphic patches. Ghost patches can utilize randomization to create polymorphic patches that can be distributed based on different heuristics (i.e. based on region, Operating System version, or staggered by time). The non-deterministic nature of a polymorphic ghost patch could make exploit development more difficult because the same patch would not be applied to each end system. In this case, the traditional patch would also have to be altered for each patch instance to prevent attackers who utilize multiple instances of a patch to expose the legitimate vulnerability.

Based on our observations, traditional patches for input validation vulnerabilities detect malicious input and *return gracefully* from the function. This prevents a compromise, but when viewing a binary diff, searching for differences that add *return* commands could be an identification technique. Applying obfuscation to fake and legitimate patches or to the function being patched could increase the difficulty in distinguishing between each type of patch. Future work should explore obfuscation techniques to make code more difficult to understand [108] and control flow more difficult to evaluate [109].

Active Response Patches Based on the *non-interfering* property, faux patches should not alter the semantics of the program. The verify step will expose that fake patches do not alter program behavior. Thus, at worst, a brute force approach could expose the vulnerability by analyzing program behavior for each path in a program and identifying which path changes a program's behavior.

One solution is to use the active response technique for legitimate patches. Active response patches prevent a vulnerability from being exploited but respond to exploits using the same response as an unpatched program. The response could return sanitized data from the actual machine or transfer execution to a honeypot environment [77]. This masking would increase the resources necessary for dynamic analysis tools to identify unpatched systems. Further research could develop techniques that hinder or prevent exploit verification. An overview of active response patches is provided in Chapter 3.

Approach Limitation Another limitation revealed by our experiments suggests that ghost patches only have a dynamic analysis impact when there are multiple *store* operations within a program's intermediate representation (i.e. operations that includes an = sign). Programs that use standard functions (i.e. *memmov*,*memcpy*) to assign values semantically perform the same operation but are represented differently syntactically, and thus, a fake patch cannot be applied.

Adding new lines of code also could add unexpected vulnerabilities. The faux patch code is like any other code that could have a vulnerability. Ghost patched code could also be attacked. Providing attackers with additional paths that could be attacked could result in a denial of service type of attack that slows overall program runtime, which could impact the machine's performance.

During our analysis, we discovered a number of interesting side effects of faux patches. The first is that because of the use of randomized values, some faux patches are not executed by the symbolic execution engine. This could mean the random value falls outside of the symbolically modeled variable or that the symbolic execution engine selected values do not interact with the inserted faux patch. This suggests that the value used in the faux patch conditional statement should be carefully assigned depending on the domain of the associated variable.

In general, software architecture deception's effectiveness is limited because a deceptive technique does not hide real or show false deceptive behavior. Thus, given a patch, the behavior of a program with the patch applied and without the patch will be different. Given a deceptive patch where the software architecture is altered and an unpatched system, each will exhibit different behavior. Because of this, attackers could still develop an exploit given a deceptive software architecture. One way to defeat this is to also add deceptive techniques to the software input and output values. This could hide real behavior or show false behavior of a patch, which is discussed in more detail in Chapter 3.

We only analyze our proof-of-concept using one symbolic execution implementation. Symbolic execution has a number of challenges that are difficult to model. Representing memory addresses symbolically when their value is calculated based on user input is one challenge that KLEE does not directly address [110]. Future work could analyze faux patched programs using additional symbolic execution tools [111].

Finally, attackers could develop heuristics to contain this path explosion problem. Identifying patches that do not alter a program's data flow could help to expose faux patches and reduce the amount of work for an attacker. Dissimulating this information could provide a way to make these heuristics difficult to identify and apply, raising the bar for attackers to distinguish between legitimate and fake patches.

Our proof of concept implementation shows that the application of deception, in the form of fake patches, to software patching is feasible. Our evaluation shows that a faux patch does have an impact on exploit generation, increasing the number of branches in a program, by increasing the resources necessary to analyze a program. These same patches also impact a program's runtime, but this effect is not statistically significant. This suggests that deception can be used to make exploit generation using patches more resource intensive, enhancing the security of software patches. With additional research and testing, this approach, either as a stand-alone technique or in conjunction with other deceptive and detection methods, could impose an exponential increase in program analysis, making exploit generation based on patches an expensive operation while only adding a minimal increase in program runtime. Our proof of concept implemented and analyzed in this research supports this claim.

4.7 Chapter Summary

This work proposed, implemented and evaluated ghost patching as a technique to mislead attackers using patches to develop exploits against input validation vulnerabilities. We discuss fake patch properties as well as analyze a proof of concept using LLVM. Through experimentation, we found that fake patches add latency to program runtime that is not statistically significant while adding a statistically significant amount of latency to program analysis. If used by program developers as they develop patches for security flaws, we believe faux patches could disrupt the exploit generation process, providing more time for end users to update their systems.

5. DECEPTIVE DISPATCHER: COMBINING DECEPTIVE SYSTEM ARCHITECTURE WITH DEPLOY AND INSTALL CHAIN

Applying deception to the system architecture of a patch can influence attackers by causing uncertainty about the location and functionality of a patch. This chapter explores how software diversification, an MTD technique, can be applied to the current software security patching protocol. MTD techniques apply to software security patches because these patches can be implemented differently while performing the same functionality. Part of implementing these software diversified patches includes altering the deploy and install chain notifications for a patch.

This chapter also discusses the application of deception to the language used in these notifications. A general overview of how language can influence biases is also provided. A methodology is also described that adds deception to the current software patch lifecycyle by combining deceptive system architecture with deploy and install chain notifications. Applying software diversity to patch development, deceptive language to patch notifications, and re-releasing these patches as new updates can influence attackers by causing uncertainty in the reconnaissance phase of their attack. An empirical analysis of how these re-released patches could be perceived by attackers and discussion about the metrics that can be used to trigger a re-release are also provided.[1]

[1]Sections of this chapter are from our published work: Offensive Deception in Computing [9]

5.1 Example Application of MTD to Software Security Patches

We discuss a variety of ways to apply MTD to software security patches, showing how the application is a one-off approach. This suggests that the applications can be implemented using current patching protocols. Examples of MTD patches are provided and a methodology for creating and releasing diversified patches is presented in this chapter.

5.1.1 Deceptive Command Line Tools

In the Windows operating system, users can display the patches that have been applied on their system via command line tools such as *wmic*. The command *rpm* can be used to list patches on Linux systems. Applying deception to alter these commands could alter their output to reflect the presence of a patch when that patch is not present and vice versa. This serves as an example of how currently available tools and commands can be altered to respond deceptively to queries. A challenge that must be addressed, which is outside the scope of this dissertation, is how to distinguish between legitimate versus malicious use of these altered commands to report the correct information for each scenario.

5.2 Deploy and Install Chain

5.2.1 Overview

Empirically, there are three main stages where notifications occur once a patch is released. The first identifies that a patch is available. This notification can be presented through an update platform, through an alert system where a link is provided to the update, or through email. The next phase where notifications are observable occurs when a patch is being installed. These notifications provide feedback to the end user installing the patch. For example, these notifications could include information about files that are being added, edited and/or removed, progress of the installation, restarting the computer, and prompts for users to accept or decline. The final notification identifies whether the patch was successfully installed or if an error occurred and the patch was not able to install. This message is sometimes presented to the user, or it can be found in a centralized notification center.

Background The background of this work is rooted in deceptive semantics and communication. This area has received some attention in the fields of sociology and psychology, but little has been done in the field of security. This looks at exposing biases in end users by simulating communication or by hiding real communication.

Other work has looked at examples of benevolent deception. These techniques are used to hide unnecessary or extremely technical details and/or provide relief to end users. With patches, benevolent deception is applied with the progress bar. The bar is meant to simulate the relative amount of work that has been completed by the executable to install the patch. This progress bar, though, is not an accurate or actual representation of the amount of work and is meant to give end users a sense of work being done [53].

Deceptive Text and Bias Prior work by Pfleeger et al. has studied behavioral science and its impact on cyber security tool development [112]. Ding et al. research how to create a dictionary of words from phishing emails that elicit biases [113].

Attackers use deceptive techniques to exploit end users' biases and cause them to take/not take actions that further the success of the attack. Deceptive attacks are comprised of at least one of the following components: force or fool. The force component attempts to command the recipient to follow some action. The fool component attempts to hide the deception so that it is not obvious to a recipient. Exploiting biases that appeal to emotions/triggers helps to hide elements that would expose the deception.

When a deceptive attack is viewed or received, the recipient must decide what his/her plan of action will be. This decision-making process is influenced by the words used in the attack, visual stimuli, current external factors and prior knowledge. One formal treatment of the decision-making process is the OODA loop [102]. Disrupting this process prevents an informed decision from being made. Force words interrupt this process by limiting the time available to make a decision, temporarily withholding access to something of value to the recipient, or completely removing access

Throughout this work, we use the following definition of bias by Bennett et al: An inclination to judge others or interpret situations based on a personal and oftentimes unreasonable point of view [114]. Almeshekah et. al provide an overview of bias and its role in deception [115].

Using deceptive patches exploits an attacker's *automation* bias and *anchoring or localism* bias. Attackers rely on the automatically released patch being legitimate and the vulnerabilities it fixes being present in unpatched code. These patches can support intelligence gathering by recording commands executed by unsuspecting attackers in honeypot environments as well as waste attacker resources. Prior work by Araujo et. al. [77, 78] supports using deceptive patches to improve system defense.

Deceptive Semantic Generation Because notifications and alerts are written in text and statically presented, using advertising techniques to deceive users is a viable approach. Text color, position, size, images, time of appearance, frequency, repetition, etc. can all be used to deceive those observing.

5.3 Deceptive Dispatcher

The release of a traditional patch serves as a trigger for attackers to investigate the associated program for vulnerabilities and develop a corresponding exploit. Research and enterprise defenses focus on the speed and efficiency of patching systems while they are in use [11, 116, 117]. Research on adding deception into the current software security protocol is scarce. Software diversity as an MTD deceptive technique with deploy and install chain deception can be combined to generate and release deceptive patches using current technologies that cast uncertainty on this trigger. This section describes the application of deception to a general software security patching protocol by re-releasing diversified versions of previously released patches. Chapter 2 provides details on the current patch lifecycle, software diversity and software diversity's application to patching.

5.3.1 Overview

Given a patch that was released at some prior time, t, this methodology suggests releasing a diversified or a refactored version of the same patch at time, $t + \delta$. This refactored patch will use the same or similar notifications in the deploy and install chain as the initial patch, but the code will look and behave slightly differently with the same output as the original patch. The main point of this approach is that the re-released patch addresses the same vulnerability but replaces P_O. Thus, the program's state of security remains consistent, but the code changes. The deception takes advantage of the expectation that patches change code in applications to address exploitable vulnerabilities present in software. The premise of this approach suggests that an attacker's exploit generation process will be influenced by diversified patches, causing them to search for a vulnerability that ideally has already been patched by installing the prior release.

5.3.2 Software Security Re-release Protocol

A software security patching protocol is a generalization of the series of steps that are taken to identify a vulnerability, generate, and release a patch. This protocol is based on the general patch lifecycle shown in Figure 2.1 with more granular stages.

1. **Identify vulnerability:** [43]
   (a) 3rd party identifies vulnerability and notifies vendor
   (b) Internal developer(s) identify vulnerability and notify corresponding developer
   (c) Developer of the software identifies vulnerability
2. **Replicate unintended behavior:** This step verifies the vulnerability is reachable and a security flaw.
3. **Identify approach to fix flaw:** Developers discuss how to fix the flaw and review options.
4. **Implement fix:** The actual code to fix the vulnerability is implemented
5. **Test and review the fix to verify completeness and accuracy:** The code to fix the vulnerability is reviewed by other developers
6. **Generate executable patch:** The patch is packaged such that it is ready to release and install on end user machines
7. **Release patch to public**

This general protocol identifies the major steps in the security patch protocol where these patches are for programs used by public end users. We add deception by appending steps to the end of this protocol.

8. **Document executable that was released, vulnerability fixed and notification released:** The patch executable, vulnerability and notification are saved in a database.

9. **Develop multiple diversified versions of original patch P_O:** Based on P_O, diversified versions of the patch are automatically or manually developed.

10. **Test and review diversified patch P_n:** Diversified patches are tested and reviewed by other developers to verify that they fix the original vulnerability that was addressed by P_O, have "enough diversity" when compared to P_O, and they are compatible with the program using unit tests. Once verified, the subsequent diversified patch is uploaded to the same database as P_O in the same record as the original patch.

11. **Stimulus occurs that triggers patch re-release:** After some amount of time passes, an event or series of events occur that trigger a re-release.

12. **Identify patch P_n to re-release:** The specific diversified patch to be re-released is selected based on observed stimuli.

13. **Generate executable patch:** The re-released patch is packaged such that it can be installed automatically on end user machines.

14. **Re-release patch to public**

This protocol adds deception to the patching lifecycle by re-releasing patches. These patches are diversified versions of the original patch. Not all patches can be diversified, so not all patches can be re-released. We discuss metrics to re-release a patch as well as identify candidates for re-releasing a patch. Because the re-released patch is a diversified version of the original patch, it will not alter the code's behavior given that the prior patch was installed. The patch may change other aspects of the program, so the patch may be larger than P_O. Developing these additional patches during time δ does not increase the time to release a patch. Also this protocol only duplicates the existing security in place by the original patch release P_O. Subsequent patches do not remove security from the program. We visually represent the protocol in Figure 5.1.

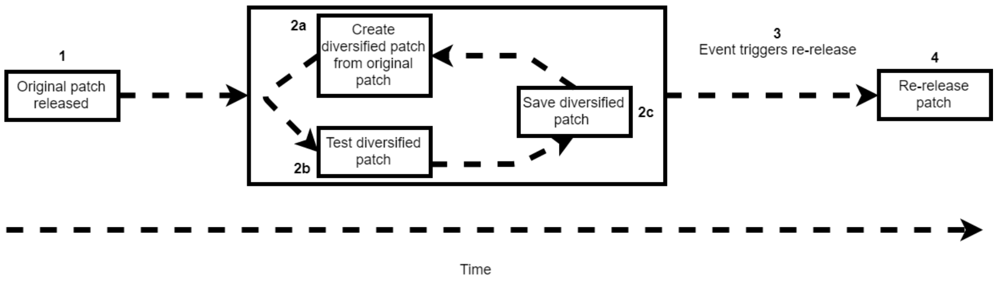

Fig. 5.1.: High Level Overview of Re-release Protocol

Patch re-release takes advantage of the expectation that patches fix flaws that exist in unpatched code. Fake patches are another approach to addressing this expectation, but the challenge with fake patches is attackers have the capability to identify the fallacy behind these patches. Chapter 4 provides more details explaining how fake patches can be distinguished from legitimate patches. Because patch re-releases are diversified versions of actual patches, they also fix the original vulnerability that was present in the unpatched code. This approach forces an attacker to identify whether a released patch is an original patch that fixes an original vulnerability or whether it is a diversified re-release that still addresses the original vulnerability. Because end users have ideally applied the original patch, it does not actually alter any program behavior. This added step increases the resources an attacker would need to expend to determine if a patch is worth exploring.

As a side effect, re-releases provide end users who have not patched their systems with P_O, the opportunity to patch the original vulnerability with a new patch. Currently, to back patch a system, the end user must identify the missed patch to be applied from an archive of prior patches or wait for a subsequent patch that may include the missed patch as part of the update. Finding the original patch could increase the workload of end users, further deterring them from installing the fix and leaving their system vulnerable to attack. Waiting for a subsequent patch also leaves the system vulnerable to attack.

The re-released patch also will have a notification announcing a new patch is available. This notification could be a duplicate of the information sent with P_O, or it could be semantically equivalent but syntactically different. Adding deceptive language and appearance to the patch deploy and install chain could influence attackers and cause them to expend resources on exploiting a vulnerability that has already been potentially fixed in end users' code.

5.3.3 Realizing a Deceptive Dispatcher

Re-releasing patches can be manually or automatically generated by the software update centers used by different operating systems. Operating systems have software update centers that receive data when an update is available and display notifications to end users about the availability of a patch. This section describes how the patch update centers can provide diversified patches, identifies metrics that can be used to trigger the release of a diversified patch, and discusses the potential security and performance impact of diversified patches.

Patch Selection Metrics Characteristics of patches cause them to be more suitable to diversify and re-released. We use this section to identify metrics that can be used to identify viable patches to diversify.

- Length of a Patch

  The length of a patch can be used to select patches to diversify. Patches comprised of more lines of code provide more data to diversify than patches with fewer lines of code. If a patch has fewer lines of code, the original version of the patch could be similar to the diversified, thus making identifying the re-release trivial. The length of a patch is a viable metric to select patches to diversify.

- Vulnerability Being Fixed

  The vulnerability a patch is fixing can dictate if a patch should be diversified. If a vulnerability can only be fixed in a limited number of ways, the goal of diversification and re-releasing patches may be compromised as the patch could be identified as a re-release. If a vulnerability can be fixed in a variety of ways, the patch for that vulnerability could be a good candidate for diversification and re-release.

Diversification Metrics Once a patch is selected to diversify, selecting the diversified version of the patch to save and potentially re-release is dependant on the difference between the each version of the patch. For example, a diversified version of a patch should be different compared to the original version of the patch such that the two cannot be trivially identified as equivalent. Also, subsequent diversified versions should also be different compared to each other for the same reason. Measuring the difference between two versions of code can be based the following metrics. Prior work has studied methods to measure the differences between obfuscated and diversified code and the results from these can influence the generation of a metric of deception [64, 65, 93, 118].

- Differences in the lines of code

  The length, lines of code, of a patch compared to the length of a diversified patch can indicate the amount of deception added as well as provide insight into the increase in workload to analyze a patch. The longer a patch, the more lines of code to analyze and/or execute, thus, the more time necessary to reverse engineer a patch. Thus, the greater the difference between the number of lines in the original patch compared to the diversified patch can be used to select a diversified version to save for re-release.

  A limitation with this metric is the greater the difference between the two versions, the larger the diversified patch and the longer the execution time of the diversified patch. Thus, performance and patch size must be considered when considering diversified patches to save for re-release.

- Number of different lines between deceptive and original patch

  The differences in the lines of code between versions of a patch can also provide a measure for the amount of deception added and identify diversified patches to save for re-release. If each line of code in a diversified patch is different than each line in an original patch, then it is more difficult to distinguish between an entirely new patch being released and a diversified patch being re-released. The differences between lines of code within each version could be caused by rearranging the code or by using different lines of code.

- Number of differences in code execution

  Another metric for the amount of diversification is the differences in system input and output between the original patch and the diversified patch. Each version of the patch can fix the same vulnerability, but respond to input and provide output differently. The number of different responses to input and different outputs could identify amount of diversification applied and make distinguishing between a re-released and a newly released patch more difficult.

Re-release Metrics Patches can be re-released for a variety of reasons. We use this section to identify a number of metrics that can be used to trigger a patch re-release. These metrics are associated with creating a campaign for attackers to follow and believe. The more plausible developers make the campaign, the more effective the deception.

- Outside/3rd Party Trigger

  Outside vendors, especially those who have partnerships or whose products are used with another vendor's products can cause a patch re-release. If a 3rd party vendor discovers a vulnerability and releases an update to their software, a separate patch might be necessary to remain compatible with the updated software. Thus, using a 3rd party's release event is a viable trigger for re-releasing a patch.

- Time

  Time could trigger a patch re-release. If substantial time, which could vary case by case, has passed between the original patch, P_O, and a diversified patch, P_n, then a re-release could be triggered.

- Attacks against software

  Discovering numerous exploits that are active against an application could trigger a re-release. When attention is on a particular program,re-releasing a patch could divert attackers' attention to develop new exploits based on the newest release.

Discussion: Side Effects and Limitations From an end user's perspective, side effects are positive. As end users, code has already been updated. Thus, re-applying a patch to code that has already been patched has minimal adverse effects. If the patch has not been applied, then there is a positive side effect if end users apply this new patch. They will have another chance to protect their systems, something that does not happen in the current patching ecosystem. The question to ask with this approach is if an end user does not apply the original patch, what probability is there that s/he will apply these subsequent patches? A user study should be conducted to determine patching behavior among end users, separating them based on different demographics. This study is outside of the scope this dissertation.

From an attacker's perspective, side effects are related to additional work. An attacker must distinguish an original patch fixing a legitimate vulnerability from a re-released patch that also fixes the same vulnerability. Ideally, the time between original and re-release will allow for the original patch to be forgotten or at least not be readily available for an attacker to compare against.

Another side effect is from a software input/output perspective. Because the re-released patch and original patch are alternative versions of the same patch, their software input and output are the same. Given a program that has been patched with P_O and a program patched with P_n, the program behavior is identical. Thus, from a user perspective, the dynamic analysis of each type of program would be identical. If the program has been kept up to date with patches, then the re-released patched program and the unpatched program, which is the originally patched program, will also have the same behavior. Given the expectation that program behavior of a patched and unpatched program should be different and based on the notification accompanying the re-released patch, viewing no change in program behavior could influence attacker behavior. A limitation of the deceptive dispatcher is that generating diversified patches expends developer's time and increases the time to release legitimate patches. Automated software diversification is an active area of research that removes the manual effort necessary to generate diversified patches. Also, diversified patches are developed once the exploitable vulnerability has been addressed. Thus, programs are no longer vulnerable when the diversified patch is being generated. Generating these patches also does not take precedent over generating patches for exploitable vulnerabilities in software.

Another challenge is that because end users do not always patch their systems, the potential for them to apply re-released patches is also limited. Developers cannot force end users to apply a patch for their software. Future work could research and identify methods that can increase the probability of patch installation. End users could be deceived by deceptive deploy and install chain messages. This could also contribute to them not installing a patch for example because of miss-understanding the criticality of the patch based on the notification.

A potential negative impact of diverse patching is an increase in patching size. Diverse patches behave identically, but use different instructions. Thus, patches that perform identically could have different sizes. This suggests that diversified patches addressing the same vulnerability could replace, edit or add one line of code or replace the entire program. Correlated to size, diversified patches could negatively affect overall program performance. Less optimized code that behaves the same but performs additional instructions compared to the original patch would execute slower. A threshold of acceptable runtimes for a program would provide developers the necessary data to test if the performance of a diversified patch is acceptable.

5.4 Chapter Summary

In this chapter, a discussion of software diversification and its application to the current software security patching protocol is presented. This chapter also discusses the application of deception to the language used in these notifications and gives a general overview of biases that are influenced based on the presence of deception. Also a methodology is described that adds deception to the current software patch lifecycle by combining deceptive system architecture with deploy and install chain notifications. Applying software diversity to patch development, deceptive language to patch notifications and re-releasing these patches as new updates can influence attackers by causing uncertainty in the reconnaissance phase of their attack. An empirical analysis of how these re-released patches could be perceived by attackers and discussion about the metrics that can be used to trigger a re-release are also provided. Finally, limitations of this deceptive dispatcher are identified.

6. FORMAL MODEL OF DECEPTIVE PATCH EFFECT

6.1 Deceptive Patch Formal Model Approach

The paradigm that deceptive data, tools and behavior are difficult to distinguish from their legitimate counterparts is intuitively understood, but a formal representation of why deception works and its impact on the security provided by patches is lacking. As an emerging research area, it is important to develop this strong foundation from which to reason about the security impact of proposed techniques.

Thus, we present a number of deceptive models that represent a variety of deceptive patches to move toward a formal model of deception. These models identify theoretically secure techniques as well as those that fall short of theoretical security. For techniques that fall short, additional analysis shows they could still be effective in practice.

In this chapter, we first introduce formal game-based security definitions that capture the technique's claimed security impact and present a general game-based model using these definitions. We then apply this general model to faux, obfuscated, and active response patches to formally analyze their security impact. Finally, we discuss whether these ideal properties of deceptive systems can be achieved in reality.

6.2 Modeling Impact of Deception on Patch-based Exploit Generation

In game-based security proofs, the probability of an adversary succeeding in a game is bounded to demonstrate the construction possesses a particular property. In simulation-based security proofs, the real-world construction is demonstrated to be

Table 6.1.: Deceptive Patch Examples

Original	**Faux**

`int vul_func(char *input_string,`
`    int input_length){`
`    char string[20]`
 	
`    strcpy(string, input_string);`
 	
`    return 0;`
`}`
	`int vul_func(char *input_string,`
	`    int input_length){`
	`    char string[20]`
	`    if(input_string == "test")`
	`        string[9] = "W"`
	`    ...`
	`    return 0;`
	`}`

Obfuscated	**Active Response**

`int vul_func(char *input_string,`
`    int input_length){`
`    char string[20];`
`    int i;`
`    while(i < input_len)`
`        string[i] = 0`
`        string[i] += input_len[i]`
`    return 0;`
`}`
	`int vul_func(char *input_string,`
	`    int input_length){`
	`    char string[20]`
	`    if(input_len > 20 || input_len < 0)`
	`        transfer_exec();`
	`    else`
	`        strncpy(...);`
	`    ...`
	`    return 0;`
	`}`

computationally indistinguishable from an ideal-world construction. Because both *ghost* and *obfuscated* patches alter the semantics of a program (the transitional patch component removes the vulnerability, thus the semantics are altered), there is no indistinguishability between the ideal and real-world solution, making a simulation-based proof trivial. Thus, in this work, we adopt the game-based approach.

6.2.1 Security Parameters

It is common to require that an adversary, $\mathcal{A}$, bound to probabilistic polynomial time (PPT) has at most a negligible advantage in breaking the security guarantee under consideration with respect to a security parameter, λ. For example, λ may be the size of the cryptographic key. In Section 6.3.3, we introduce a notion of λ in the context of deceptive patches.

6.2.2 Oracle Sampling

Both game-based and simulation-based approaches to modeling security for deceptive patching require an oracle, $\mathcal{O}$, that samples patches from a given distribution, $\mathcal{D}$. Sampling patches efficiently from $\mathcal{D}$ is less straightforward than, e.g., sampling from $\mathbb{Z}_p^*$. In particular, patches must at minimum retain a degree of plausibility to prevent an adversary from constructing an efficient distinguisher.

6.2.3 Complexity of Adversary Actions

The PPT adversary, $\mathcal{A}$, tends to have two primary operations: using `Identify` to locate the vulnerability a patch fixes, and using `Verify` to check that the vulnerability is exploitable in unpatched systems. In general we argue that both of these have straightforward polynomial time constructions for most deceptive patching techniques. However, we shall see an example of a deceptive technique where `Verify` may not allow the construction of an efficient distinguisher.

6.2.4 Game Assumptions

We assume characteristics about a game's sets and adversary capabilities.

- We assume both deceptive and legitimate patch sets are large. These sets must be large to allow adversaries to query for examples as well as provide a challenge an unknown number of times.
- We assume an adversaries has unlimited resources. We do not place time bounds on an adversary.
- We assume the set of deceptive and legitimate patches contain similar content.
- We assume the set of deceptive and legitimate patches are of similar size.
- We assume an adversary can use any analysis technique to study the oracle's response.
- We assume active responses can mimic any response from a legitimate system.

6.2.5 Generalized Game-based Model of Deceptive Patch Impact

The generalized game-based model of deceptive patch impact identifies the protocol used to represent the effect of deceptive patches.

1. Adversary, $\mathcal{A}$, requests a polynomial number of patches, `P`, sampled from a set of legitimate patches, $\mathcal{L}$, and deceptive patches, $\mathcal{D}$.

2. The Oracle responds with random `P` $\xleftarrow{\$} \mathcal{L}$ and `P` $\xleftarrow{\$} \mathcal{D}$. The adversary can request patches individually from each set, with full knowledge of the set from which the patch originates.

3. $\mathcal{A}$ requests a challenge response, `P'`.

4. The system uniformly selects a random value $b \in \{0,1\}$, which determines whether the response is sampled from $\mathcal{L}$ or $\mathcal{D}$. Thus, $\mathcal{A}$ must distinguish whether `P'` is a legitimate or deceptive patch and potentially identify the vulnerability being addressed.

5. Optionally, adversary, $\mathcal{A}$, requests a polynomial number of patches, `P`, sampled from a set of legitimate patches, $\mathcal{L}$, and deceptive patches, $\mathcal{D}$.

6. Again, the Oracle responds with random `P` $\xleftarrow{\$} \mathcal{L}$ and `P` $\xleftarrow{\$} \mathcal{D}$. The adversary can request patches individually from each set, with full knowledge of the set from which the patch originates. These responses are mutually exclusive from all prior responses and an adversary cannot specifically request to see the challenge response.

7. Eventually, $\mathcal{A}$ outputs a guess bit $\mathbf{b}' \in \{0,1\}$, and wins whenever $\mathbf{b}' = b$ and loses otherwise.

We apply the general protocol above to faux, obfuscated, and active response patches. This protocol provides insight into the impact of deceptive patches. An adversary achieving greater than 50% plus some trivial percentage suggests that the enhancements to security by the deceptive patch under analysis are distinguishable.

6.2.6 Faux Patches

Adversarial Model

We consider a PPT adversary, $\mathcal{A}$, that attempts to distinguish between a legitimate patch and a *faux* patch. We assume that $\mathcal{A}$ has access to and can interact fully with samples of both legitimate and faux patches, as both are generally publicly available.

Indistinguishability Game

In the Faux Patch Indistinguishability game, an adversary $\mathcal{A}$ is asked to distinguish between a patch, $\texttt{P}$, sampled from legitimate, ($\texttt{P} \in \mathcal{L}$), or faux, ($\texttt{P} \in \mathcal{F}$), patches.

Protocol 6.2.1: $\texttt{P}$−IND Patch Indistinguishability

	Adversary $\mathcal{A}$		Patch Oracle $\mathcal{O}$	
(1)	Request $\texttt{P} \in \mathcal{L}$, $0 \le i \le \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$\texttt{P} \overset{\$}{\leftarrow} \mathcal{L}$, $0 \le i \le \text{poly}(\lambda)$	**(2)**
(3)	Request $\texttt{P} \in \mathcal{F}$, $0 \le i \le \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$\texttt{P} \overset{\$}{\leftarrow} \mathcal{F}$, $0 \le i \le \text{poly}(\lambda)$	**(4)**
(5)	Request Challenge	$\longrightarrow$	$b \in \{0, 1\}$	
		$\longleftarrow$	$b(\texttt{P'} \overset{\$}{\leftarrow} \mathcal{L}) + (1-b)(\texttt{P'} \overset{\$}{\leftarrow} \mathcal{F})$	**(6)**
(7)[1]	Request $\texttt{P} \in \mathcal{L}, \texttt{P} \neq \texttt{P'}$, $0 \le i \le \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$\texttt{P} \in \mathcal{L}, 0 \le i \le \text{poly}(\lambda)$	**(8)**
(9)	Request $\texttt{P} \in \mathcal{F}, \texttt{P} \neq \texttt{P'}$, $0 \le i \le \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$\texttt{P} \in \mathcal{F}$, $0 \le i \le \text{poly}(\lambda)$	**(10)**
(11)	Guess $\mathbf{b'} \overset{?}{=} b$	$\longrightarrow$		

The Faux Patch Indistinguishability game of Protocol 6.2.1 proceeds as follows: (1) Adversary, $\mathcal{A}$, requests a polynomial number of patches `P` sampled from the set of legitimate patches, $\mathcal{L}$. (2) The Oracle responds with random `P` $\xleftarrow{\$} \mathcal{L}$. (3) Similarly, $\mathcal{A}$ requests a polynomial number of patches `P` sampled from the set of faux patches $\mathcal{F}$. (4) The Oracle responds with random `P` $\xleftarrow{\$} \mathcal{F}$. (5) $\mathcal{A}$ requests a challenge patch, `P'`. (6) The system uniformly selects a random value $b \in \{0, 1\}$, which determines whether the patch is sampled from $\mathcal{L}$ or $\mathcal{F}$. Thus, $\mathcal{A}$ must distinguish whether `P'` is a legitimate or faux patch. (7) Adversary, $\mathcal{A}$, optionally requests a polynomial number of legitimate patches `P` $\xleftarrow{\$} \mathcal{L}$ that have not been queried before, and such that `P` $=$ `P'`. (8) The Oracle responds with sampled patches `P` $\xleftarrow{\$} \mathcal{L}$. (9) Similarly, $\mathcal{A}$ optionally requests a polynomial number of faux patches `P` $\xleftarrow{\$} \mathcal{F}$ that have not been queried before, and such that `P` $=$ `P'`. (10) The Oracle responds with sampled patches `P` $\xleftarrow{\$} \mathcal{F}$. (11) Eventually, $\mathcal{A}$ outputs a guess bit $\mathbf{b}' \in \{0, 1\}$, and wins whenever $\mathbf{b}' = b$ and loses otherwise.

Let $\text{Adv}_{\mathcal{A}}^{\mathcal{F}-\text{IND}} = Pr[\mathbf{b}' = b]$ represent the probability of $\mathcal{A}$ winning the game. We require that the advantage of a PPT adversary, $\mathcal{A}$, is $|\text{Adv}_{\mathcal{A}}^{\mathcal{F}-\text{IND}} - \frac{1}{2}| \leq \epsilon$ where ϵ is a negligible function in the security parameter, λ.

6.2.7 Obfuscated Patches

Adversarial Model

We consider an adversary, $\mathcal{A}$, bound to PPT that attempts to identify whether the obfuscated response is an *obfuscated* patch fixing a hidden underlying vulnerability or just obfuscated code with no underlying vulnerability. We assume that $\mathcal{A}$ has

[1]In game-based security models, the adversary is allowed to continue querying the oracle after receiving the challenge on any input which is not the challenge itself. This permits adaptive adversaries, who use knowledge of the challenge to influence their strategy [119].

access to the obfuscated response. $\mathcal{A}$ also can interact with the obfuscated response to validate whether the vulnerability they have identified exists.

Vulnerability Identification Game

In the Obfuscated Patch Identification game, an adversary, $\mathcal{A}$, is asked to identify whether obfuscated code is a patch ($\mathtt{P} \in \mathcal{O}$) fixing an underlying vulnerability, $\mathcal{V}$, or just obfuscated code with no underlying vulnerability ($\bar{\mathtt{P}} \in \mathcal{O}$).

The Vulnerability Identification game of Protocol 6.2.2 proceeds as follows: (1) Adversary, $\mathcal{A}$, requests a polynomial number of patches, $\mathtt{P}$, sampled from the set of obfuscated code, $\mathcal{O}$. (2) The oracle responds with random $\mathtt{P} \xleftarrow{\$} \mathcal{O}$. (3) $\mathcal{A}$ attempts to identify the vulnerability obfuscated in each patch, $\mathtt{P}$. (4) $\mathcal{A}$ attempts to validate the legitimacy of the identified vulnerability obfuscated in each patch, $\mathtt{P}$. (5) Adversary, $\mathcal{A}$, requests a polynomial number of non-patched code, $\bar{\mathtt{P}}$, sampled from the set of obfuscated code, $\mathcal{O}$. (6) The oracle responds with random $\bar{\mathtt{P}} \xleftarrow{\$} \mathcal{O}$. (7) $\mathcal{A}$ attempts to validate the lack of a vulnerability in the obfuscated non-patched code, $\bar{\mathtt{P}}$. (8) $\mathcal{A}$ requests a challenge, $\mathtt{c}$, such that $\mathtt{c}$ has not been seen in prior requests nor will be seen in subsequent requests. (9) Optionally, adversary, $\mathcal{A}$, requests a polynomial number of patches, $\mathtt{P}'$, sampled from the set of obfuscated code, $\mathcal{O}$, that have not been requested previously. (10) The oracle responds with random $\mathtt{P}' \xleftarrow{\$} \mathcal{O}$. (11) $\mathcal{A}$ attempts to identify the vulnerability obfuscated in each patch, $\mathtt{P}'$. (12) $\mathcal{A}$ attempts to validate the legitimacy of the identified vulnerability obfuscated in each patch, $\mathtt{P}'$. (13) Optionally, adversary, $\mathcal{A}$, requests a polynomial number of non-patched code, $\bar{\mathtt{P}'}$, sampled from the set of obfuscated code, $\mathcal{O}$, that have not been previously requested. (14) The oracle responds with random $\bar{\mathtt{P}'} \xleftarrow{\$} \mathcal{O}$. (15) $\mathcal{A}$ attempts to validate the lack of a vulnerability in the obfuscated non-patched code, $\bar{\mathtt{P}'}$. (16) Eventually, $\mathcal{A}$ outputs a guess bit $\mathbf{b}' \in \{0, 1\}$, and wins whenever $\mathbf{b}' = b$ and loses otherwise.

Protocol 6.2.2: $\mathcal{V}-$ID Vulnerability Identification

	Adversary $\mathcal{A}$		Patch Oracle $\mathcal{O}$	
(1)	Request $\texttt{P} \in \mathcal{O}$, $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$\texttt{P} \overset{\$}{\leftarrow} \mathcal{O}$, $0 \leq i \leq \text{poly}(\lambda)$	**(2)**
(3)	$\texttt{Identify}(\mathcal{V} \in \texttt{P})$, $0 \leq i \leq \text{poly}(\lambda)$			
(4)	$\texttt{Verify}(\mathcal{V} \in \texttt{P})$, $0 \leq i \leq \text{poly}(\lambda)$			
(5)	Request $\bar{\texttt{P}} \in \mathcal{O}$, $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$\bar{\texttt{P}} \overset{\$}{\leftarrow} \mathcal{O}$, $0 \leq i \leq \text{poly}(\lambda)$	**(6)**
(7)	$\texttt{Verify}(\mathcal{V} \notin \bar{\texttt{P}})$, $0 \leq i \leq \text{poly}(\lambda)$			
(8)	Challenge $\mathbf{c} \notin \texttt{P}, \bar{\texttt{P}}, \texttt{P}', \bar{\texttt{P}'}$	$\longrightarrow$	$b \in \{0,1\}$	
		$\longleftarrow$	$\mathbf{r} \leftarrow b(\texttt{P}(\mathbf{c})) + (1-b)(\bar{\texttt{P}}(\mathbf{c}))$	**(6)**
(9)	Request $\texttt{P'} \in \mathcal{O}, \texttt{P'} = \texttt{P}$ $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$\texttt{P'} \overset{\$}{\leftarrow} \mathcal{O}$, $0 \leq i \leq \text{poly}(\lambda)$	**(10)**
(11)	$\texttt{Identify}(\mathcal{V} \in \texttt{P'})$, $0 \leq i \leq \text{poly}(\lambda)$			
(12)	$\texttt{Verify}(\mathcal{V} \in \texttt{P'})$, $0 \leq i \leq \text{poly}(\lambda)$			
(13)	Request $\bar{\texttt{P}'} \in \mathcal{O}, \bar{\texttt{P}'} = \bar{\texttt{P}}$ $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$\bar{\texttt{P}'} \overset{\$}{\leftarrow} \mathcal{O}$, $0 \leq i \leq \text{poly}(\lambda)$	**(14)**
(15)	$\texttt{Verify}(\mathcal{V} \notin \bar{\texttt{P}'})$, $0 \leq i \leq \text{poly}(\lambda)$			
(16)	Guess $\mathbf{b'} \overset{?}{=} b$	$\longrightarrow$		

Let $\text{Adv}_{\mathcal{A}}^{\mathcal{V}-\text{ID}} = Pr[b = 1]$ represent the probability of $\mathcal{A}$ winning the game. We require that the advantage of a PPT adversary, $\mathcal{A}$, is $|\text{Adv}_{\mathcal{A}}^{\mathcal{V}-\text{ID}}| \leq \epsilon$ where ϵ is negligible in the security parameter, λ.

6.2.8 Active Response Patches

Adversarial Model

We assume an adversary, $\mathcal{A}$, bound to PPT that interacts with a remote system, $\mathcal{S}$, in a black box manner. That is, $\mathcal{A}$ exchanges messages with $\mathcal{S}$ and attempts to distinguish with non-negligible advantage between two possible states of $\mathcal{S}$: a patched state, $\mathtt{P}$, or an unpatched state, $\bar{\mathtt{P}}$. In the case of deceptive and obfuscated patches, we assume that $\mathcal{A}$ has access to the patched and unpatched source code, as this is generally publicly available, and can interact with each version for an unlimited amount of time.

Indistinguishability Game

In the Active Response Indistinguishability game, an adversary, $\mathcal{A}$, is asked to distinguish between a patched ($\mathtt{P}$) or unpatched ($\bar{\mathtt{P}}$) remote system, $\mathcal{S}$, by issuing challenges and evaluating the corresponding responses. Note that $\mathcal{A}$ can evaluate not only the content of the response, but also auxiliary information, `Aux` (e.g., packet delay).

The Active Response Patch Indistinguishability game of Protocol 6.2.3 proceeds as follows:

(1) Adversary, $\mathcal{A}$, issues a polynomial number of challenges c_i to a patched system, thus we denote the challenges as members of the set $\mathcal{C}'_{\mathtt{P}}$ which is a subset of all challenges $\mathcal{C}'$ issued before the distinguishing stage. (2) The system queries the patched system on c_i and returns the corresponding response $r_i \leftarrow \mathtt{P}(c_i)$. (3) Similarly, $\mathcal{A}$ issues a polynomial number of challenges c_i to an unpatched system (denoted $\bar{\mathtt{P}}$), and we denote the challenges as members of the set $\mathcal{C}'_{\bar{\mathtt{P}}}$. (4) The system queries the unpatched system on c_i and returns the corresponding response $r_i \leftarrow \bar{\mathtt{P}}(c_i)$. (5)

Protocol 6.2.3: $\mathtt{AR}$−IND Patch Indistinguishability

	Adversary $\mathcal{A}$		Server $\mathcal{S}$	
(1)	$c_i \in \mathcal{C}'_{\mathtt{P}} \subset \mathcal{C}'$, $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$r_i \leftarrow \mathtt{P}(c_i)$, $0 \leq i \leq \text{poly}(\lambda)$	**(2)**
(3)	$c_i \in \mathcal{C}'_{\bar{\mathtt{P}}} \subset \mathcal{C}'$, $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$r_i \leftarrow \bar{\mathtt{P}}(c_i)$, $0 \leq i \leq \text{poly}(\lambda)$	**(4)**
(5)	Challenge $\mathbf{c} \notin \mathcal{C}'$	$\longrightarrow$	$b \in \{0, 1\}$	
		$\longleftarrow$	$\mathbf{r} \leftarrow b(\mathtt{P}(\mathbf{c})) + (1-b)(\bar{\mathtt{P}}(\mathbf{c}))$	**(6)**
(7)	$c'_i \in \mathcal{C}_{\mathtt{P}}, \mathbf{c} = c'_i$, $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$r'_i \leftarrow \mathtt{P}(c'_i)$, $0 \leq i \leq \text{poly}(\lambda)$	**(8)**
(9)	$c'_i \in \mathcal{C}_{\bar{\mathtt{P}}}, \mathbf{c} = c'_i$, $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$r'_i \leftarrow \bar{\mathtt{P}}(c'_i)$, $0 \leq i \leq \text{poly}(\lambda)$	**(10)**
(11)	Guess $\mathbf{b'} \overset{?}{=} b$	$\longrightarrow$		

$\mathcal{A}$ selects a challenge, $\mathbf{c}$, which has not been issued previously, and queries the system. (6) The system uniformly selects a random value $b \in \{0, 1\}$, which determines whether the challenge is issued to a patched or unpatched system. Thus, $\mathcal{A}$ must distinguish whether $\mathbf{r}$ was the response from $\mathtt{P}$ or $\bar{\mathtt{P}}$. (7) Adversary, $\mathcal{A}$, optionally issues a polynomial number of challenges $c'_i \notin \mathcal{C}'$ to a patched system, such that c'_i has not been queried before. (8) The system queries the patched system on c'_i and returns the corresponding response $r'_i \leftarrow \mathtt{P}(c'_i)$. (9) Similarly, $\mathcal{A}$ optionally issues a polynomial number of challenges c'_i to an unpatched system (denoted $\bar{\mathtt{P}}$). (10) The system queries the unpatched system on c'_i and returns the corresponding response $r'_i \leftarrow \bar{\mathtt{P}}(c'_i)$. (11) Eventually, $\mathcal{A}$ outputs a guess bit $\mathbf{b}' \in \{0, 1\}$, and wins whenever $\mathbf{b}' = b$ and loses otherwise.

Let $\text{Adv}_{\mathcal{A}}^{\mathtt{AR}-\text{IND}} = Pr[\mathbf{b}' = b]$ represent the probability of $\mathcal{A}$ winning the game. We require that the advantage of a PPT adversary, $\mathcal{A}$, is $|\text{Adv}_{\mathcal{A}}^{\mathtt{AR}-\text{IND}} - \frac{1}{2}| \leq \epsilon$ where ϵ is negligible in the security parameter, λ.

6.3 Achieving the Ideal

To claim a primitive such as deceptive patching increases the security of a system, the primitive must be shown to satisfy a meaningful definition of a security property. As we will see, the assumptions that form the model of deceptive patches are unrealistic in practice for faux and obfuscated patches, suggesting that these categories are not formally secure.

6.3.1 Faux Patches

By empirical analysis, $\text{Adv}_{\mathcal{A}}^{\mathcal{F}-\text{IND}} - \frac{1}{2} \leq \epsilon$ for a negligible function ϵ will not hold true, as given access to an unpatched system the adversary can run $\mathtt{Verify}(\mathcal{V})$ in polynomial time on any vulnerability, $\mathcal{V}$, they have identified in the patch, `P`. Because adversaries have access to the code, dynamic analysis and program execution can be used to verify if a patch is faux or legitimate. Automated tools such as KLEE [106] and Triton [120] use symbolic execution to develop reliable exploits [44] and can help attackers distinguish between the patches because the response from a faux patch during execution is distinguishable from that of a legitimate patch. Faux patches should not alter data flow nor should they alter control flow by returning from a function while legitimate patches can exhibit at least one of these characteristics. Thus, in practice, the assumption of similar set content does not hold true.

Despite adding faux patches to alter the available paths, symbolic execution, or dynamic analysis in general, can run in polynomial time to verify whether a patch is faux or legitimate. As $\texttt{P} \in \mathcal{F}$, by definition $\mathcal{V}$ does not exist and so $\mathcal{A}$ has a non-negligible advantage in distinguishing patches sampled from $\mathcal{F}$ or $\mathcal{L}$.

6.3.2 Obfuscated Patches

One approach to cryptography requires formal security definitions based on the presumed difficulty of computationally bounded adversaries from solving well-studied mathematical problems. Obfuscated patching follows the latter approach, employing ad hoc methods to disguise the underlying vulnerability addressed by the patch.

Obfuscated patches are comparable to the goals of white box cryptography [121], which attempts to obfuscate keys embedded in software made available to adversaries. However, it is not known whether any rigorous security guarantees can be achieved in this model, as cryptanalysis has broken white box constructions [122].

An adversary, $\mathcal{A}$, with access to $\texttt{Verify}(\mathcal{V})$ will not have advantage $|\mathrm{Adv}_{\mathcal{A}}^{\mathcal{V}-\mathrm{ID}}| \leq \epsilon$, as both `Identify` and `Verify` run in polynomial time. Automated tools exist that perform automated exploit generation and code analysis in polynomial time, even with the application of obfuscation [33, 123]. Also, the set of obfuscated patches and obfuscated non-patch code are not similar as patches can alter data and control flow of a program, i.e. *return* statements, while non-patch code should not alter data nor control flow.

6.3.3 Active Response Patches

The goal of active response patches is to prevent an adversary from constructing a distinguisher for patched and unpatched systems which has a non-negligible advantage. We argue that this deceptive patching technique may be able to satisfy $|\mathrm{Adv}_{\mathcal{A}}^{\mathtt{AR}-\mathrm{IND}} - \frac{1}{2}| \leq \epsilon$ in Protocol 6.2.3.

The strategy of the remote system, $\mathcal{S}$, with which the adversary interacts is to design a deceptive patch that fixes the underlying vulnerability, but issues responses indistinguishable from an unpatched system (discussed in Section 6.4). Even though $\mathcal{A}$ is given access to the patch and has complete[2] information, it may not be possible to remotely distinguish the responses from either an unpatched or patched system with non-negligible advantage.

We propose using the size of the domain from which vulnerable code can respond as the security parameter λ, i.e., the space from which vulnerable code can respond to an exploit. The larger this domain, the higher the probability of a deceptive approach being distinguishable. The smaller this domain, the lower the probability of a deceptive approach being distinguishable. That is, if many responses exist from exploiting vulnerable code, a deceptive patch is less likely to be indistinguishable as an attacker has a larger surface with which to verify the legitimacy of the patch.

[2]In game theory, *complete* information refers to games where all players have complete knowledge of the game structure and payoffs. In contrast, games of *perfect* information allow all players to observe every move by other players. The distinguishing game of Protocol 6.2.3 is a game of complete but imperfect information, as $\mathcal{A}$ has access to the patch yet does not observe whether or not the system invokes the patch.

If a vulnerability only has one response to exploits against it, a deceptive patch for this vulnerability has a higher probability of being indistinguishable from unpatched systems, as the ability to verify the legitimacy of a patch is constrained.

Because active response patches meet the criteria of being theoretically secure, we define a λ for these types of patches. Active response patches attempt to mimic vulnerable machines. Thus the ability to completely mimic a vulnerable machine and how these machines will respond to exploits against a vulnerability is key to the success of these patches. Thus, λ represents the security parameter such that the impact of deception on the security provided by a patch is either directly or inversely related to the size of λ (i.e. the larger the security parameter the larger impact the deceptive patch has on program security and the smaller the security parameter smaller the impact the deceptive patch has on program security).

For active response patches, the security parameter, λ, is the size of the space that must be modeled. The size of the space can be represented by the number of responses possible for the vulnerability being patched. In general, this space can be described using a spectrum. On one end are vulnerabilities that when exploited result in a program crashing or have a single course of action. These vulnerabilities are trivial to model. On the other end, vulnerabilities that result in memory leaks, escalated privileges, etc. are more difficult to model. This difficulty stems from the fact that every capability of an attacker who successfully exploits a vulnerability that displays or impacts contents in memory is unknown a priori. Thus, the more possibilities an active response must model, the less secure the algorithm. The fewer possibilities an active response must model, the more secure the algorithm.

6.3.4 General Security Parameter

There is no general security parameter of deceptive patches. The security parameter is a measure of the difficulty to distinguish between legitimate and deceptive patches. The length and/or size of the security parameter provides an indication about the difficulty of distinguishing between a deceptive and legitimate patch. Faux and obfuscating patches attempt to increase the distance between the software architecture, system input and output, system architecture and/or deploy and install chain of these deceptive patches and legitimate patches.

There is no general security parameter that provides insight as to the difficulty of distinguishing a deceptive patch or choosing a better deceptive patch over a moderate deceptive patch. The intuition is based on the fact that some deceptive patches alter code to hide the real patch, i.e. faux and obfuscation patches, and others alter code to show the false patch to make it seem real, i.e. active response patches. Deceptive patches that alter system architecture attempt to make changes to the code that it is different from the nondeceptive version. These deceptive techniques attempt to create a deceptive patch that does not appear the same as a legitimate patch by hiding the real elements using noise or rearranging code. Creating these different versions makes the original/legitimate patch more difficult to identify because the software architecture or system input and output is different from the expected. Thus, the security parameter for patches that hide the real elements of a patch should be based on a measure of difference between the original patch and the faux or obfuscated patch. Developing a general measure of this difference is outside the scope of this research, but prior work has identified techniques to measure differences between versions of code [64]. Active response patches attempt to appear the same as legitimate patches in that the system input and output data are indistinguishable. Because these patches show false information that appears real, the security parameter should be based on how similar responses from active response patches are to vulnerable systems. Thus, the security parameter for active response patches indicates a similarity between deceptive and legitimate patches while the security parameter for faux and obfuscation patches indicates a differences between the two types of patches.

6.4 Realizing Active Response Patches

We have argued that active response patches may be able to satisfy a meaningful formal security definition. We now present plausible methods of implementing and deploying active response mechanisms which satisfy the security definition.

Active responses provide deceptive data to attackers in real time. That is, data is dynamically or statically generated and presented to the attacker to influence their decision making process. We discuss two techniques for implementing active responses in software security patches, as well as advantages and challenges of each.

6.4.1 Virtual Machine Implementation

Some websites use threads and/or virtual machines (VMs) to provide clients with content. Clients are sandboxed in their own VM or thread as their requests are analyzed and delivered. A VM or thread based infrastructure is well-suited to implement active response patches that transfer execution to virtualized honeypot environments. The vulnerable VM or thread will be isolated and instrumented such that an adversaries actions can be stealthily monitored, allowing the defender to learn information about the adversary's strategy and goals.

Motivation

The goal of invoking VMs is to transfer execution to an isolated and sandboxed environment upon detection of an adversary attempting to compromise a machine through a patched exploit. Transferring execution to a honeypot environment allows the operational software to continue servicing legitimate requests, and the attack steps may be monitored and logged without impacting operational machine performance and security. This could aid security analysis, and defenders identify novel attack vectors while ensuring legitimate data remains safe.

Framework

This approach is composed of two phases: the detection phase, and the deception and monitoring phase. The detection phase identifies the exploit, while the deception and monitoring phase begins when the attacker executes commands within the VM. The VM should be vulnerable to the same vulnerability that triggered the transfer of execution. The VM is populated with deceptive and potentially legitimate data. Thus, when attackers execute commands, the VM responds with data that plausibly exists on the operational machine being attacked. As attackers execute commands and access data, their activities are logged for future analysis.

Challenges

Using a honeypot to transfer execution once an exploit is detected introduces unique challenges to system security. The honeypot must be periodically updated with plausible and active data as out-of-date files and login information could expose the sandbox [124]. The data must appear plausible, as attackers could have access to other data and techniques to verify the data, causing them to mistrust the information

[80]. These VMs could also be detected by the adversary through timing analysis, and identifying the presence or absence of hardware and drivers [125]. Another challenge is preventing attackers from overloading the honeypots and potentially using them to launch attacks against the legitimate system.

Because prior work shows sandbox environments can be fingerprinted consistently and reliably by an adversary [124], this approach would be best applied to environments where *all* execution (both regular and honeypot) is performed in virtual machines that are started when a user requests resources and stopped when they are finished.

Advantages

This approach creates an interactive and isolated environment that can be efficiently controlled by defenders. Defenders can monitor activity within VMs and quickly create, restart and stop VM's compared to physical machines. This flexibility gives defenders the ability to adapt to attackers' methods.

6.4.2 Non-Virtual Machine Implementation

Some computing environments allow users to remotely access the physical hardware. These systems could use virtualization to transfer execution to a vulnerable sandboxed VM when an exploit is detected. Prior research has shown that an adversary can distinguish between sandbox environments and normal user machines [124]. Thus, other approaches to implementing active response patches that do not use the sandbox technique must be explored.

Motivation

Using deceptive defense on the local machine through the use of a deceptive daemon could influence how attackers execute their attack. Keeping both the data and the attacker stationary and on the same machine removes the need for back-end data to be duplicated in the case of the honeypot implementation. Thus, dynamic and plausible deceptive responses can be presented using real time data.

Framework

The deceptive daemon is comprised of two phases: detection stage and monitoring stage. The detection stage occurs when an attack is launched against a previously patched flaw. This detection code will identify an active exploit and invoke the deceptive daemon. The monitoring stage occurs when the daemon has been invoked for a specific flaw. This stage observes the process that initialized the exploit and monitors all execution. Active responses are presented to attackers in this phase, and during monitoring the daemon will determine how to respond to requests. Once a process has been identified as initiating an attack, that process is considered tainted. Every request from and response to these processes must be identified, analyzed, and deceptive techniques applied based on security policies. Responses to requests will be cached along with the process requesting the data and "kill chain" events to form a signature of the attacker [7]. This signature can be used to identify an adversary in subsequent attacks. The cached data can be used to quickly respond to similar requests, as well as preserve consistency across multiple exploits against the same flaw [126]. Execution can be transferred to a honeypot during an exploit if an executable is uploaded to isolate its execution.

Challenges

This approach introduces unique challenges to secure and protect a host machine, as the daemon runs on the operational machine along with legitimate programs and data. Challenges about storage must also be addressed, as deceptive data and policy statements are cached and saved. Trade-offs between storing the data on disk or using an external device must be measured. This approach will also impact system performance, as the daemon will consume system resources that are traditionally reserved for legitimate processes. Running the program on the machine in a stealthy manner is important to keep attackers engaged, as the daemon being discovered may also be a deterrent. Having techniques and methods in place if the daemon crashes must be addressed. Using redundancy by implementing multiple versions of deceptive daemons on a machine would keep the machine protected in case of a failure.

Advantages

This approach prevents data that is not related to an exploit from needlessly being copied. Only the data that is necessary is identified and processed.

6.5 Chapter Summary

In this chapter, we present a generic template to model the theoretical security of deceptive patches. This template can be used to identify the decisions an attacker will make when provided deceptive content. We also show how some deceptive patches do not provide theoretical security, but suggest that these patches may still have benefit to defenders and developers. We also prove that deceptive patches that alter patch behavior provide theoretical security within our security model.

7. CONCLUSIONS AND FUTURE WORK

Deception adds a layer of defense to current defense techniques. Software security patches are the state of practice in defending against vulnerabilities. These updates prevent exploits from succeeding by blocking the vulnerability. This means that malicious code that would take advantage of the flaw in an unpatched program is prevented from successfully exploiting the program. This occurs by either removing malicious content through error correcting or exiting the program itself. Deception is an added layer of defense that, used with traditional preventative techniques, can enhance the security of systems.

7.1 Business Case

Deceptive patches can be applied to different corporate/business categories. Certain technologies could implement some active response patches, but given the current state of obfuscation and faux patching, these are not feasible to apply with great confidence to successfully thwart attackers. Because active response patches are indistinguishable from unpatched systems, an attacker in the reconnaissance phase performing dynamic analysis only interacts with the system's responses. Businesses with technology that uses remote servers that respond to requests can employ active response patches to better protect their systems by identifying information that interests attackers. By applying active response patches to vulnerabilities that can be exploited, defenders can gain information about attacker's interests as he/she performs reconnaissance to begin their attack and interacts with active response patches. Businesses with remote servers such as Google, Microsoft, and others with websites that could contain vulnerabilities could benefit by applying active response patches in their applications. Businesses that use, support or develop tools that take user input without the user viewing the code being executed can apply active response patches.

Results in this dissertation suggest there is promise for implementing faux patches to increase attacker workload. The results suggest that an attacker's workload increases when analyzing patches to develop exploits. Further studies must be conducted to identify additional program characteristics where faux patch application is advantageous. Businesses that release patches to end users to download could apply faux and/or obfuscated patches to increase the analysis time of attackers developing exploits based on patches.

7.2 Summary

In this dissertation, we identified and presented how deception can be applied to software security patches. First, we discussed patch-based exploit generation, the motivation for our work. We discussed current events that fall under this type of attack. We discussed the literature providing background material for deceptive patches as well as prior work in the area of deceptive patching. We then identified and discussed the major components of a software patch in Chapter 3. We discussed how these components can be deceptively implemented as well as how adding deception to each component impacts the cyber kill chain. We also presented a timeline analysis of the effects of deceptive patches.

Prior work has looked at manually adding deceptive patches to code, but our explanation of ghost patches in Chapter 4 is the first to add fake input validation patches to code using an automated compiler tool. We discuss implementation, analysis and implications of this work. Results suggest deceptive patching is feasible. We implemented, tested and analyzed a tool that inserts deceptive patches, specifically fake patches into code. We show that for programs where an input variable determines the number of times a loop executes, either directly or indirectly, the number of paths in a faux patched program that are enumerated by KLEE is unequal to the number of paths in unpatched programs. We did not observe any general characteristics in the KLEE-benchmark suite that indicated an addition of faux patches would increase the number of paths enumerated. We also performed an experiment that added multiple conditional statements for each store instruction in the program. This experiment showed that increasing the number of faux patches that are inserted per store instruction does not generally increase the number of paths nor the runtime of KLEE to enumerate these paths. The experiment had the opposite effect in some instances, decreasing the number of paths and decreasing KLEE's runtime compared the paths and runtime after adding a single faux patch per store instruction. We conclude that adding faux patches to programs does not generally increase the number of paths or runtime of KLEE. This suggests that adding faux patches does not induce a path explosion when using symbolic execution to analyze a program. Thus, faux patches do not generally increase the dynamic analysis of a program nor increase the workload of an attacker. Our experiments suggest that other factors such as the values used in the conditional statements of faux patches and the comparison operators used influence the number of paths more than the number of faux patches added.

This chapter presents two measures of the difficulty of deception, specifically in analyzing deception. The runtime of an analysis tool as well as the number of paths throughout a program indicate the amount of deception added as well as the difficulty to analyze and study this deception. The runtime of a program provides an indication to the difficulty of analyzing a program using brute force techniques. If the runtime of an analysis tool increases when analyzing deceptive patches, this suggests that the amount of data to analyze overall has increased. The number of paths throughout a program provide a similar indication to the change in workload for an attacker. Optimizations have been implemented within KLEE, i.e. dead path and dead code removal, such that using this tool is applied to model a relatively knowledgeable attacker.

In Chapter 5, we discuss the idea of introducing moving target defense techniques to software security patches and provide analysis based on prior work in the semantics of deception on software security patch notifications. Finally, we present a framework using the traditional software patching lifecycle, add subsequent steps to generate diverse versions of released patches, and discuss metrics that trigger the release of these diversified patches.

In Chapter 6 we analyzed a formal model of deceptive patches that examines the impact of deceptive patches using a game theoretic approach.

7.3 Future Work

The research conducted and presented in this dissertation provides a number of results that can be used to continue to progress the field of deceptive patching. Our compiler approach is the first to apply deceptive techniques to software security patching using automated techniques.

One additional piece of work to extend the implementation of faux patches applies the technique to additional vulnerability classes. Inserting fake patches that appear to fix cross-site scripting vulnerabilities, buffer overflows, and other string related vulnerabilities provides an interesting area of study and expands the capabilities of faux patches. Also, performing a user study to measure the distinguishability between faux patches and legitimate patches would provide insight on additional characteristics that must be present to make faux patches more plausible. Performing this user study with both participants who are knowledgeable about computer science, coding and exploits and those who are naive would provide interesting results.

Another interesting area of research is to develop more comprehensive testing benchmarks for deceptive tools. One component that is lacking wide range support in the area of deceptive patching is testing and specifically how deceptive tools can be tested and shown to be effective as well as efficient. Developing benchmarks and baseline measurements so that researchers can more effectively gauge the influence of their tools is key to progressing this field of research.

Researching and applying machine learning to deception is a future area of study that could have an impact on the way we perform defense. Classification and clustering techniques can be used to identify and develop the efficient and effective deceptive patch given inputs such as the vulnerability being fixed, the length of time the vulnerability has been public, the size of the project, etc. Truly automated patch development and application and then deceptive patch development and application guides us toward automated software security where applications are able to harden themselves against exploit.

As research on deceptive patches expands, new proposed techniques should be evaluated with respect to a clear and meaningful definition of security. The shift to rigorous modeling transitioned cryptography from an art to a science, and this approach should be followed by other areas claiming security guarantees.

This dissertation presents components and a general workflow for a Deceptive Dispatcher tool that re-releases diversified versions of previously released patches. Implementing and analyzing the performance and effectiveness of this should guide future work.

REFERENCES

REFERENCES

[1] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "Opus: Online patches and updates for security," in *Usenix Security*, vol. 5, 2005, p. 18.

[2] Symantec, "Internet security threat report," 2015.

[3] V. Enterprise and Affiliates, "2015 data breach investigations report," 2015.

[4] M. Kumar, "Wannacry ransomware: Everything you need to know immediately," 2017, http://thehackernews.com/2017/05/how-to-wannacry-ransomware.html.

[5] T. Rains, "When vulnerabilities are exploited: The timing of first known exploits for remote code execution vulnerabilities," Microsoft, Tech. Rep., 2014.

[6] F. Cohen, "A note on the role of deception in information protection," *Computers & Security*, vol. 17, no. 6, pp. 483–506, 1998.

[7] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, p. 80, 2011.

[8] J. Avery and E. H. Spafford, "Ghost patches: Fake patches for fake vulnerabilities," in *International Conference on ICT Systems Security and Privacy Protection.* IEEE, 2017.

[9] J. Avery, M. Almeshekah, and E. Spafford, "Offensive deception in computing," in *International Conference on Cyber Warfare and Security*, 2017, p. 23.

[10] "Patch (computing)," 2016, https://en.wikipedia.org/wiki/Patch_(computing).

[11] M. Payer and T. R. Gross, "Hot-patching a web server: A case study of asap code repair," in *International Conference on Privacy, Security and Trust.* IEEE, 2013, pp. 143–150.

[12] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto, "Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing," in *Security & Privacy.* IEEE, 2007, pp. 252–266.

[13] D. Zamboni, "Using internal sensors for computer intrusion detection," Ph.D. dissertation, Purdue University, 2001.

[14] B. Brykczynski and R. A. Small, "Reducing internet-based intrusions: Effective security patch management," *Software*, vol. 20, no. 1, pp. 50–57, 2003.

[15] J. Corbet, "How to participate in the linux community," 2008, http://ldn.linuxfoundation.org/book/how-participate-linux-community.

[16] A. Arora, J. P. Caulkins, and R. Telang, "Research note—sell first, fix later: Impact of patching on software quality," *Management Science*, vol. 52, no. 3, pp. 465–471, 2006.

[17] "Patch management," 2008, https://www.infosec.gov.hk/english/technical/files/patch.pdf.

[18] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack, "Timing the application of security patches for optimal uptime." in *Large Installation System Administration Conference*, vol. 2, 2002, pp. 233–242.

[19] J. Dadzie, "Understanding software patching," *ACM Queue*, vol. 3, no. 2, pp. 24–30, 2005.

[20] M. K. McKusick, "A conversation with Tim Marsland." *ACM Queue*, vol. 3, no. 4, pp. 20–28, 2005.

[21] A. Arora, R. Krishnan, R. Telang, and Y. Yang, "An empirical analysis of software vendors' patching behavior: Impact of vulnerability disclosure," *International Conference on Information Systems*, p. 22, 2006.

[22] J. Dadzie, "Understanding software patching," *ACM Queue*, vol. 3, no. 2, pp. 24–30, 2005.

[23] J.-W. Sohn and J. Ryoo, "Securing web applications with better "patches": An architectural approach for systematic input validation with security patterns," in *International Conference on Availability, Reliability and Security.* IEEE, 2015, pp. 486–492.

[24] M. Monperrus, "A critical review of automatic patch generation learned from human-written patches: Essay on the problem statement and the evaluation of automatic software repair," in *International Conference on Software Engineering.* ACM, 2014, pp. 234–242.

[25] T. Wang, C. Song, and W. Lee, "Diagnosis and emergency patch generation for integer overflow exploits," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2014, pp. 255–275.

[26] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *Security & Privacy.* IEEE, 2012, pp. 48–62.

[27] I. V. Krsul, "Software vulnerability analysis," Ph.D. dissertation, Purdue University, 1998.

[28] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *USENIX Security Symposium*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006, http://dl.acm.org/citation.cfm?id=1267336.1267349.

[29] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *SIGCOMM Workshop on Large-Scale Attack Defense.* ACM, 2006, pp. 131–138.

[30] S. Rugaber, T. Shikano, and R. K. Stirewalt, "Adequate reverse engineering," in *International Conference on Automated Software Engineering.* IEEE, 2001, pp. 232–241.

[31] M. Popa, "Binary code disassembly for reverse engineering," *Journal of Mobile, Embedded and Distributed Systems*, vol. 4, no. 4, pp. 233–248, 2012.

[32] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Working Conference on Reverse Engineering.* IEEE, 2002, pp. 45–54.

[33] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Working Conference on Reverse Engineering.* IEEE, 2005, pp. 10–pp.

[34] C. Wang and S. Suo, *The Practical Defending of Malicious Reverse Engineering.* University of Gothenburg, 2015.

[35] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *International Conference on Software Engineering.* IEEE Press, 2012, pp. 771–781.

[36] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *SIGCOMM Workshop on Large-Scale Attack Defense.* ACM, 2006, pp. 131–138.

[37] T. Rains, "When vulnerabilities are exploited: The timing of first known exploits for remote code execution vulnerabilities," *Microsoft Secure Blog*, 2014, https://blogs.microsoft.com/microsoftsecure/2014/06/17/when-vulnerabilities-are-exploited-the-timing-of-first-known-exploits-for-remote-code-execution-vulnerabilities/.

[38] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Security & Privacy*, May 2008, pp. 143–157.

[39] C. Percival, "Binary diff/patch utility," 2003, http://www.daemonology.net/bsdiff.

[40] H. Flake, "Structural comparison of executable objects," in *Detection of Intrusions and Malware & Vulnerability Assessment*, 2004, pp. 161–173, http://subs.emis.de/LNI/Proceedings/Proceedings46/article2970.html.

[41] B. Lee, "Darungrim: A patch analysis and binary diffing tool," *Black Hat*, 2011.

[42] J. Oh, "Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries," *Black Hat*, 2009.

[43] L. Ablon and A. Bogart, *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits.* Rand Corporation, 2017.

[44] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation." in *Network and Distributed System Security Symposium*, vol. 11, 2011, pp. 59–66.

[45] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security & Privacy*. IEEE Computer Society, 2010, pp. 317–331.

[46] A. Dewdey, *Computer Recreations, a Core War Bestiary of Virus, Worms and Other Threats To Computer Memories*. Scientific America, 1985, vol. 252.

[47] K. D. Mitnick and W. L. Simon, *The Art of Deception: Controlling the Human Element of Security*. John Wiley & Sons, 2011.

[48] E. Spafford, "More than passive defense," 2011, https://www.cerias.purdue.edu/site/blog/post/more_than_passive_defense/.

[49] C. Stoll, *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Simon and Schuster, 2005.

[50] M. B. Salem and S. J. Stolfo, "Decoy document deployment for effective masquerade attack detection," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2011, pp. 35–54.

[51] J. Yuill, M. Zappe, D. Denning, and F. Feer, "Honeyfiles: Deceptive files for intrusion detection," in *SMC Information Assurance Workshop*. IEEE, 2004, pp. 116–122.

[52] F. Cohen *et al.*, "The deception toolkit," *Risks Digest*, vol. 19, 1998.

[53] E. Adar, D. S. Tan, and J. Teevan, "Benevolent deception in human computer interaction," in *SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013, pp. 1863–1872.

[54] M. H. Almeshekah and E. H. Spafford, "Using deceptive information in computer security defenses," *International Journal of Cyber Warfare and Terrorism*, vol. 4, no. 3, pp. 46–58, 2014.

[55] N. Rowe, "A taxonomy of deception in cyberspace," in *International Conference on Information Warfare and Security*, 2006, pp. 173–181.

[56] U. States, *Military Deception*, ser. Joint pub 3-13.4. Joint Chiefs of Staff, 2006, https://fas.org/irp/doddir/dod/jp3_13_4.pdf.

[57] U. S. of America, *Joint Doctrine for Military Deception*, ser. Joint pub 3-58. Joint Chiefs of Staff, 1996, http://purl.access.gpo.gov/GPO/LPS50196.

[58] F. Cohen, D. Lambert, C. Preston, N. Berry, C. Stewart, and E. Thomas, "A framework for deception," *National Security Issues in Science, Law, and Technology*, 2001.

[59] J. J. Yuill, "Defensive computer-security deception operations: Processes, principles and techniques," Ph.D. dissertation, North Carolina State University, 2006.

[60] M. H. Almeshekah and E. H. Spafford, "Planning and integrating deception into computer security defenses," in *New Security Paradigms Workshop*. ACM, 2014, pp. 127–138.

[61] J. B. Bell and B. Whaley, *Cheating and Deception.* New Brunswick, N.J. : Transaction Publishers, 1991, reprint, with new introd. Originally published: Cheating. New York, N.Y. : St. Martin's Press, 1982.

[62] B. Whaley, "Toward a general theory of deception," *The Journal of Strategic Studies*, vol. 5, no. 1, pp. 178–192, 1982.

[63] C. Collberg and J. Nagra, "Surreptitious software," *Upper Saddle River, NJ: Addision-Wesley Professional*, 2010.

[64] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The University of Auckland, Tech. Rep., 1997.

[65] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Security & Privacy.* IEEE, 2014, pp. 276–291.

[66] J. Xu, P. Guo, M. Zhao, R. F. Erbacher, M. Zhu, and P. Liu, "Comparing different moving target defense techniques," in *Workshop on Moving Target Defense.* ACM, 2014, pp. 97–107.

[67] B. Coppens, B. D. Sutter, and K. D. Bosschere, "Protecting your software updates," *Security & Privacy*, vol. 11, no. 2, pp. 47–54, 2013.

[68] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security.* Springer, 2008, pp. 100–120.

[69] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *New Security Paradigms Workshop.* ACM, 2010, pp. 7–16.

[70] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo, "Baiting inside attackers using decoy documents," in *Security and Privacy in Communication Networks*, 2009, pp. 51–70.

[71] B. Whitham, "Canary files: Generating fake files to detect critical data loss from complex computer networks," in *Cyber Security, Cyber Peacefare and Digital Forensic.* The Society of Digital Information and Wireless Communication, 2013, pp. 170–179.

[72] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *SIGSAC Conference on Computer & Communications Security.* ACM, 2013, pp. 145–160.

[73] C. Laney, S. O. Kaasa, E. K. Morris, S. R. Berkowitz, D. M. Bernstein, and E. F. Loftus, "The red herring technique: A methodological response to the problem of demand characteristics," *Psychological Research*, vol. 72, no. 4, pp. 362–375, 2008.

[74] M. A. Bashar, G. Krishnan, M. G. Kuhn, E. H. Spafford, and S. Wäğstäff Jr, "Low-threat security patches and tools," in *International Conference on Software Maintenance.* IEEE, 1997, pp. 306–313.

[75] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Workshop on Digital Rights Management.* Springer, 2001, pp. 160–175.

[76] I. Balepin, S. Maltsev, J. Rowe, and K. Levitt, "Using specification-based intrusion detection for automated response," in *International Workshop on Recent Advances in Intrusion Detection.* Springer, 2003, pp. 136–154.

[77] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," in *SIGSAC Conference on Computer and Communications Security.* ACM, 2014, pp. 942–953.

[78] F. Araujo, M. Shapouri, S. Pandey, and K. Hamlen, "Experiences with honey-patching in active cyber security education," in *Workshop on Cyber Security Experimentation and Test*, 2015.

[79] T. Holz and F. Raynal, "Detecting honeypots and other suspicious environments," in *SMC Information Assurance Workshop.* IEEE, 2005, pp. 29–36.

[80] K. E. Heckman, M. J. Walsh, F. J. Stech, T. A. O'Boyle, S. R. DiCato, and A. F. Herber, "Active cyber defense with denial and deception: A cyber-wargame experiment," *Computers & Security*, vol. 37, pp. 72–77, 2013.

[81] S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Booby trapping software," in *New Security Paradigms Workshop.* ACM, 2013, pp. 95–106.

[82] J. Heusser and P. Malacaria, "Quantifying information leaks in software," in *Annual Computer Security Applications Conference.* ACM, 2010, pp. 261–269.

[83] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi, "Packet vaccine: Black-box exploit detection and signature generation," in *Conference on Computer and Communications Security.* ACM, 2006, pp. 37–46.

[84] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense.* Springer, 2011, pp. 77–98.

[85] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[86] S. Banescu, M. Ochoa, and A. Pretschner, "A framework for measuring software obfuscation resilience against automated attacks," in *International Workshop on Software Protection.* IEEE, 2015, pp. 45–51.

[87] D. Dunaev and L. Lengyel, "Method of software obfuscation using petri nets," in *Central European Conference on Information and Intelligent Systems.* Faculty of Organization and Informatics Varazdin, 2013, p. 242.

[88] Y. Kanzaki, A. Monden, and C. Collberg, "Code artificiality: A metric for the code stealth based on an n-gram model," in *International Workshop on Software Protection.* IEEE Press, 2015, pp. 31–37.

[89] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Symposium on Principles of Programming Languages.* ACM, 1998, pp. 184–196.

[90] J. N. Stewart, "Advanced technologies/tactics techniques, procedures: Closing the attack window, and thresholds for reporting and containment," *Best Practices in Computer Network Defense: Incident Detection and Response*, vol. 35, p. 30, 2014.

[91] H. C. Goh, "Intrusion deception in defense of computer systems," Master's thesis, Naval Postgraduate School, 2007.

[92] I. Balepin, S. Maltsev, J. Rowe, and K. Levitt, "Using specification-based intrusion detection for automated response," in *International Workshop on Recent Advances in Intrusion Detection.* Springer, 2003, pp. 136–154.

[93] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense," in *Workshop on Moving Target Defense.* ACM, 2014, pp. 31–40.

[94] A. Cui and S. J. Stolfo, "Symbiotes and defensive mutualism: Moving target defense," in *Moving Target Defense.* Springer, 2011, pp. 99–108.

[95] R. Zhuang, S. Zhang, A. Bardas, S. A. DeLoach, X. Ou, and A. Singhal, "Investigating the application of moving target defenses to network security," in *International Symposium on Resilient Control Systems.* IEEE, 2013, pp. 162–169.

[96] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *Security & Privacy*, vol. 12, no. 2, pp. 16–26, 2014.

[97] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: Transparent moving target defense using software defined networking," in *Workshop on Hot Topics in Software Defined Networks.* ACM, 2012, pp. 127–132.

[98] D. Cohen, "Surrogate indicators and deception in advertising," *The Journal of Marketing*, pp. 10–15, 1972.

[99] D. M. Gardner, "Deception in advertising: A conceptual approach," *The Journal of Marketing*, pp. 40–46, 1975.

[100] A. Arora, R. Krishnan, R. Telang, and Y. Yang, "An empirical analysis of software vendors' patching behavior: Impact of vulnerability disclosure," *International Conference on Information Systems*, p. 22, 2006.

[101] H. Chang, "Building self-protecting software with active and passive defenses," Ph.D. dissertation, Purdue University, 2003.

[102] J. R. Boyd, "The essence of winning and losing," 1996.

[103] J. Bambenek, "The patch window is gone: Automated patch-based exploit generation," *Infosec Community Forum*, 2005, https://isc.sans.edu/forums/diary/The+Patch+Window+is+Gone+Automated+PatchBased+Exploit+Generation/4310/.

[104] H. Cavusoglu, H. Cavusoglu, and J. Zhang, "Economics of security patch management." in *Workshop on the Economics of Information Security*, 2006.

[105] C. Lattner, *The LLVM Compiler Infrastructure.* University of Illinois, Urbana-Campaign, 2017, http://llvm.org/.

[106] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Operating Systems Design and Implementation*, vol. 8, 2008, pp. 209–224.

[107] C. Lattner, “LLVM and Clang: Next generation compiler technology,” in *The BSD Conference*, 2008, pp. 1–2.

[108] S. Dolan, “mov is turing-complete,” pp. 1–4, 2013, http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf.

[109] A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” in *Applied Computer Security Applications Conference.* IEEE, 2007, pp. 421–430.

[110] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *Security & Privacy.* IEEE, 2010, pp. 317–331.

[111] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[112] S. L. Pfleeger and D. D. Caputo, “Leveraging behavioral science to mitigate cyber security risk,” *Computers & Security*, vol. 31, no. 4, pp. 597–611, 2012.

[113] K. Ding, N. Pantic, Y. Lu, S. Manna, and M. I. Husain, “Towards building a word similarity dictionary for personality bias classification of phishing email contents,” in *International Conference on Semantic Computing.* IEEE, 2015, pp. 252–259.

[114] M. Bennett and E. Waltz, *Counterdeception Principles and Applications for National Security.* Artech House Norwood, MA, 2007.

[115] M. H. Almeshekah and E. H. Spafford, “Cyber security deception,” in *Cyber Deception.* Springer, 2016, pp. 25–52.

[116] I. Lee, “Dymos: A dynamic modification system,” Ph.D. dissertation, University of Wisconsin, Madison, 1983.

[117] G. Buban, P. Donlan, A. Marinescu, T. McGuire, D. Probert, H. Vo, and Z. Wang, “Patching of in-use functions on a running computer system,” Patent 7 784 044, Aug. 24, 2010.

[118] Y. Kanzaki, A. Monden, and C. Collberg, “Code artificiality: A metric for the code stealth based on an n-gram model,” in *International Workshop on Software Protection.* IEEE Press, 2015, pp. 31–37.

[119] O. Goldreich, *Foundations of Cryptography.* New York, NY, USA: Cambridge University Press, 2006, vol. 1.

[120] F. Saudel and J. Salwan, “Triton: A dynamic symbolic execution framework,” in *Symposium sur la sécurité des technologies de l’information et des communications.* SSTIC, 2015, pp. 31–54.

[121] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot, "A white-box des implementation for drm applications," in *Workshop on Digital Rights Management*, Springer. ACM, 2002, pp. 1–15.

[122] W. Michiels, P. Gorissen, and H. D. L. Hollmann, "Cryptanalysis of a generic class of white-box implementations," in *Selected Areas in Cryptography*, 2008, pp. 414–428, https://doi.org/10.1007/978-3-642-04159-4_27.

[123] S. Banescu, M. Ochoa, and A. Pretschner, "A framework for measuring software obfuscation resilience against automated attacks," in *International Workshop on Software Protection (SPRO).* IEEE, 2015, pp. 45–51.

[124] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes *et al.*, "Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion," in *International Symposium on Research in Attacks, Intrusions, and Defenses.* Springer, 2016, pp. 165–187.

[125] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: Vmm detection myths and realities," in *USENIX Workshop on Hot Topics in Operating Systems.* USENIX Association, 2007, pp. 6:1–6:6, http://static.usenix.org/legacy/events/hotos07/tech/full_papers/garfinkel/garfinkel_html/.

[126] M. H. Almeshekah, "Using deception to enhance security," Ph.D. dissertation, Purdue University, 2015.

VITA

VITA

Jeffrey K. Avery was born the younger of twin boys in Heidelberg, Germany in 1990. After a number of moves because of military orders, he graduated co-valedictorian from Bel Air High School, Bel Air, MD in 2008. Afterward, he graduated *magna cum laude* from the University of Maryland, Baltimore County in 2012 as a Meyerhoff Scholar with his bachelor's degree in computer science. That same year, he was admitted into the Computer Science Ph.D. program at Purdue University. He received his Master's Degree, also in Computer Science, in 2014.

Jeffrey completed a number of successful internships, including stints at Army Research Laboratory, Johns Hopkins Applied Physics Laboratory, McAfee, Inc., and Sypris Electronics, Inc. His research interests include network security, anti-phishing and deception. Upon receipt of his Ph.D., he accepted a position as a software engineer in the Future Technical Leaders program at Northorp Grumman.