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

June 2017

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

This work is dedicated to all who have had a hand in making me who I am today. I thank you with all of my heart.

ACKNOWLEDGMENTS

I cannot thank my advisor Prof. 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 you to be the research scientist I am today. The countless lessons you have taught me over the years through formal and informal conversations will serve me for the rest of my life and career. I would also like to thank my committee, Prof. Saurabh Bagchi, Prof. Christina Nita-Rotaru and Prof. Dongyen Xu for their support and advice throughout this process.

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

I would also like to 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. 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

Page

Page

Page

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 $>= 1$

ABBREVIATIONS

SDLC	Software Development Life Cycle
SOTA	State of the art
MTD	Moving Target Defense

ABSTRACT

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

Deception has been used for thousands of years to influence peoples' thoughts. Comparatively, deception has been used in computing since the 1970s. Its application to security has been documented in a variety of research studies and products on the market, but continues to evolve with new studies 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 Life-cycle (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 their 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, enhancing the security of patch.

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 attacker's decision making and exploit generation are called deceptive patches. Deceptive patching techniques could include dead-end functions, false responses to commands, and responding 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.

It is feasible to apply deception to software security patches to influence attackers' decision making. Using a variety of deceptive techniques, it is possible to develop and deploy deceptive patches.

1. INTRODUCTION

Software patching, an update to code such that there are differences between subsequent versions, is an accepted practice by end users and application developers. Delivery time constraints, third party programs, nonstandard coding practices, and other metrics contribute to bugs and vulnerabilities being introduce into programs that need to be fixed. The release of a software patch (patch or update for short) traditionally means that a vulnerability that can be exploited exists in unpatched versions of a program. This notification provides the green light for attackers to develop exploits for unpatched systems. The potential use of patches to generate malicious exploits in the wild motivates the premise of this research.

The patching ecosystem is beneficial for end users as well as malicious actors. Fixing flaws in code is advantageous for end users, but the benefit for malicious actors is captured by a Symantec Internet security threat report [1] 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 [2]. This report states that 99.9% of the exploits that were detected took advantage of vulnerabilities made public 1+ years prior [2]. In May 2017, unpatched systems were left vulnerable for months after the 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 Wan-

naCry ransomware [3]. As additional evidence of patch motivated exploits, in 2014, Tim Rains, Microsoft's Director of Security, released a blog [4] 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..." [1].

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 help protect patches and the programs they are fixing 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 actor's 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.*

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

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. Deceptive patches can cause attackers to mistrust data collected from their exploits [5], not attack a system at all to prevent wasting resources, fear being exposed, waste time attempting to develop an exploit for an incorrectly identified vulnerability. Thus, the security of deceptive patches relies on altering an attacker's approach, causing him to cast doubt on the data he collects. To enhance the security of patches, we discuss the impact of deception 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 deceptive awareness. We show the resiliency of deceptive patches to exploits where attackers know nothing about the system to complete knowledge of the 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, a vulnerability 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.

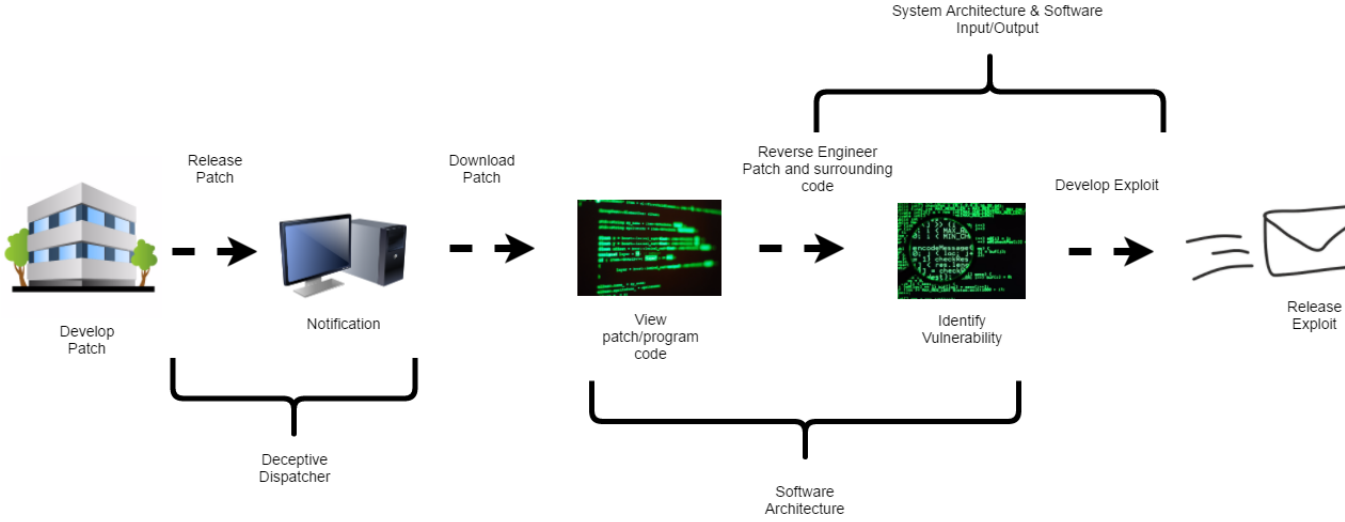

Fig. 1.1.: Patch-based exploit generation timeline with deceptive patch components/research overlay.

Figure 1.1 illustrates the patch-based exploit generation process and overlays where the chapters of 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 looks 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 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 development and maintenance lifecycle of a program is a preliminary application of deception to the Software Development Life-cycle (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 also 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 in the form of a general timeline comparing an attacker's timeline and an end user's timeline from the time a patch is released. We also map our deceptive patching model onto the cyber kill chain [6] to show how deceptive patching can effect an attacker's path to compromise. We also provide a model to analyze the security of different types of deceptive patches. We discuss a general model that can be applied to all deceptive patches and present and analyze specific examples of different deceptive patches. We discuss what types of patches can theoretically be considered as secure.

Chapter 4 discusses what makes up a patch from the architecture 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) [7].

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 describes 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. We suggest the need to distinguish between diversified versions of the same patch will increased 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) [8].

Finally, Chapter 6 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 and types of patches, economic principles that support patching and patch development. We also discuss how patches can be exploited, which significantly motivates this research and finally 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 permeates throughout the program. Wrappers can detect exploits if the exploit alters the system's state and/or program's behavior.

2.1 Patch: A definition

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*

[1]http://www.pcmag.com/encyclopedia/term/48892/patch

changed functions aggregated based on source file of their origin [9]. 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* [9].

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

A modification that closes a known vulnerability at a specific location 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.

2.1.1 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 based on who develops them can be categorized as unofficial, those developed by 3rd party vendors, or traditional, those developed by the original code developers [10]. While traditional patches require a system restart to be applied, hot patches are applied to software

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

as it is executing without the need to restart. Prior work by Payer et al. explores how dynamic analysis and sandboxes can provide patches for vulnerabilities during runtime [11]. 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. 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 or removing code within the function. This type of patch can mitigate 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 the patch release process and each tier is described in Table 2.1 [15].

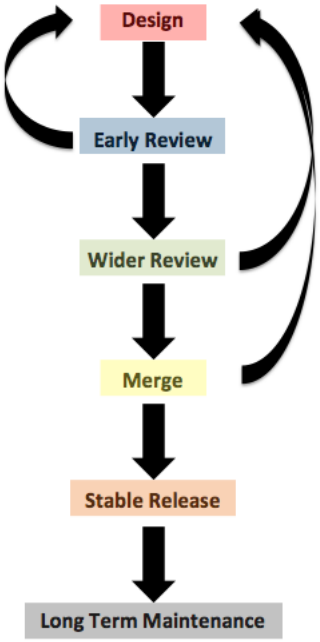

Fig. 2.1.: Life cycle of patches

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.

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	The patch is deployed to the public
Long-term maintenance	Developers maintain the patch as the code undergoes other improvements

Table 2.1.: The patching cycle

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, which represents 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 that is present in the code and 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 has been gathered as well as the minimal amount of time since public notification. Studies also suggest that public notification of a vulnerability increases the speed to patch a program [18, 21].

The software development phase an application is currently in also 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 has a different cost as opposed to code in the production or development phase.

2.4 Patch Generation

Patches are generated using manual analysis and coding or automated tools. Manual patching can be observed in practice, while automated generation has been mainly studied in academic research settings.

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. 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 [22].

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 [23]. Research by Wang et al. detects integer overflow vulnerabilities and provides a patch to fix the flaw [24].

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 [25]. Work by Krsul provides a full treatment of vulnerability analysis and categorization [26]. Work by Xie et al. uses static analysis techniques to identify vulnerabilities in software [27]. Analysis by Frei et al. uses patch and exploit release data to identify trends in vulnerability detection, exploit, and patching [28]. Deceptive patch development relies on vulnerability research to identify classes of vulnerabilities. This research aids in identifying vulnerability classes that are viable for deceptive patching.

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 applies to widening 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 [29]. 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 [30, 31]. Udupa et al. study how to reverse engineer programs deceptively treated with obfuscation techniques [32]. 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 [33]. This limited availability of "anti-reverse engineering" techniques is where deception can be used. Adding fake patches does not prevent reverse engineering from occurring, but it does alter the returned data, increasing the workload of an attacker. Future work in deceptive patches will begin to more effectively address this challenge.

2.5.3 Patched-Based Exploit Generation

Attackers use released patches to develop exploits against unpatched machines [34, 35]. This is possible because software patches inherently create a disadvantage for keeping the code being updated protected. The disadvantage is that the changes in code 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. Another disadvantage is they alter the behavior of a program, which once the patch is applied results in a more secure and hardened code, 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 drastically diminished as a change in program behavior can be used as the initial identifier that a patch is present and exactly what inputs trigger the execution of a patch [36, 37].

A specific area of exploit generation uses patches to generate attacks. Patch based exploit generation develops exploits for unpatched systems based on the patched version of a program [37, 38].

When analyzing a patched and unpatched system, a patch's semantics are used to identify differences. These differences can then be used to identify the location of a patch, what the patch does and, in turn, the vulnerability being fixed. using semantics and changes in program behavior, an attacker with access to a patched and unpatched executables of the same program could use fuzzing techniques to identify a patch and identify the vulnerability [36, 37]. Thus, even without the source code, using fuzzing techniques that interact and measure changes in program and system state, a vulnerability can be identified and an exploit developed. The basic premise of this class of exploit generation is that following new branches in a patched version of code will exploit vulnerabilities in old versions of code [38]. Because additional branches can be automatically identified between versions of code, generating classes of input that cause a program to take a new path can be done automatically.

Another form of analyzing patched and unpatched code uses the syntax differences between the two versions to locate a patch. 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 [39, 40].

2.6 Deception

Deception has been used in computing since the 1970s [41–44]. 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 [45] and honeyfiles [46] as well as the Deceptive Toolkit [47]. 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 [48, 49]. 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 [50–53].

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* [54, 55].

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 list that shows each component and how each is broken down according to [56, 57]:

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 and reason about. Introducing noise to output can make this more difficult to understand. Prior work by Collberg et al. provide a taxonomy of software obfuscation techniques [58, 59].

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 [60].

2.6.3 Deceptive Patches

Diverse Patch Applying software diversity to software security patches is a specific application of MTD techniques. Path diversity addresses the mono-culture problem created by current patching practices and could negatively impact resources used to develop and exploit. A framework presented by Coppens et al. introduces the idea of using diversification to protect patches by released different versions of the same patch to end users [61]. 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 wide spread attack [60, 62, 63]. 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. Ideally, fake patches are 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. A faux patch is best applied to input validation vulnerabilities. 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 vulnerabilities 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 [45, 46, 64, 65] and decoy passwords [66], as they are all red herring techniques [67].

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 [40, 68]. 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 [61], control flow obfuscation [58, 69], and encrypting patches [68].

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) [70]. 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 [71, 72].

Unlike the previous two deceptive techniques, active response patches do not rely on security by obscurity. The adversary is assumed to have access to the patch, and yet 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*. 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 vulnerability and others that have been intentionally left in the honey pot. Thus, the behavior of a patched and unpatched machine appears equivalent when in reality the machine is protected against exploits for that particular vulnerability [71]. The limitations with this work 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 [73] and during live exercises [74].

Crane et al. present a framework that describes how code can be instrumented to place fake instructions in locations an attack could use for an attack, but that would not be used by legitimate programs. These fake instructions would appear to perform as expected, but also send information to defenders about code being accessed [75]. The main limitation of this work is the lack of implementation and analysis that show if this technique is feasible in practice.

This dissertation advances the field of deceptive patching by analyzing a formal model, 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 created to show its potential impact on attackers. This chapter breaks the space of deceptive patching down to its most basic level of a patch, identifying four major components 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 Patches Components

The space of patches is categorized into four areas what 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 (LoC, memory size), loops, conditional statements, variables. These elements can be measured without executing a patch.

- System Input and Output : The activity of a patch. What a patch does and how it does it. The visible inputs and changes in state from a patch. These metrics can be observed while a program is executing before, during and after patch code executes.

- 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 that a patch is available. How end users are identified that a patch is available to download. Also, the information portrayed to end users during a patch's installation.

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 can be thought of as 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 include lines of code, 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 includes elements of a patch that are provided to begin patch execution or as a result of executing the patch. Any information that is input into the patch or output from the patch during an execution would fall into this category, including runtime, program state, register values, common paths and dead code. The behavior of a patch also includes information that may be leaked by executing a patch. An example is given a patch that prevents some input from reaching a certain point in code, the observation that the code stops short leaks information about the state of the program as well as the functionality of a patch [76].

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. If we think of a patch as a band-aid to close a wound, which represents a vulnerability in this metaphor, the patch can be placed in a number of different areas. The band-aid could be placed at the site of the wound but on the surface of the skin. This resembles a wrapper patch that is outside the function that is vulnerable but detects pre and post conditions [13]. The band-aid could actually be placed in the wound resembling an internal patch that is directly at the site of the vulnerability, inside the function. The patch could also be visualized as an entire sterile room where the wounded individual is placed, preventing any malicious elements from entering the vulnerable individual's environment. This resembles data patches that are implemented in firewalls and on sensors that detect malicious traffic, well before reaching any vulnerable machine [12, 77].

3.1.4 Deploy and Install Chain

Deploy chain elements of a patch include any publicly available information that a patch can be downloaded and installed. These notifications can be pushed to end user machines and displayed via notification/update protocols, sent via email to end users, or alerts provided and where the patch can be found/downloaded.

Install chain includes elements concerning installing the patch and understanding the parts that a patch will effect. 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. Patches are not just code to fix issues, but there is an entire container around patches that provides information to system administrators and end users about the state of patch installation and files that were changed, added or removed. This can also be called the patch lifecycle. The steps within this lifecycle are:

1. Notice - identification that a patch will be released or has been released and can be downloaded.

2. Release - the moment that the patch is released. This could be simultaneous to the notice.

3. Download - when an end user downloads a patch onto a machine.

4. Install - when a patch is executed to be installed on a machine.

5. Result/Outcome - whether a patch is successfully or unsuccessfully installed on a machine.

	Software Architecture	System Input and Output	System Architecture	Deploy and Install Chain
Mask	Hide architecture components	Hide I/O	Hide the location	Hide notifications
Repackage	hide the real patch within something else that is functional	active response, hide the behavior of a patching 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 looks like a real one	fake patch that behaves like a real patch	fake location that seems real;	fake notification that seems real
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

Table 3.1.: Modeling the Space of Deceptive Patching

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 influence component of deception.

3.2.1 Deceptive Software Architecture

Deceptive software architecture applies deceptive techniques to the software architecture of a patch. Applying these techniques to the physical attributes of a patch. This includes elements such as coding 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. [68] Hiding these elements does not mean that they have to truly be invisible. In order 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 can not 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 [60, 61, 78].

Dazzle Dazzling software architecture components attempts to confuse an adversary. Examples include obfuscating source code [58, 59, 79–82].

Mimic Mimicking software architecture components creates fake copies of real patches. These fake copies look, feel and even behave similarly to their real counterparts. Chapter 4 and work by Colbert et al. [58] 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.

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 act like or look like their real counterparts, but they have enough similarities such that they seem like they are real.

Chapter 5 addresses the application of decoying, mimicking, dazzling and inventing to software security patches.

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 very 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 withing 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 RED-HERRING tool [71, 72] as well as work by Crane et al. [75].

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 [83], Goh [84] and Balpin et al. [85] discuss responding to intrusions using various techniques, including deception.

Mimic Mimicking system input and output entails copying legitimate input or output and 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 like another patch. Arujo et al. [71] and Crane et al. [75] 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 fool adversaries.

Decoy Decoying system input and output copies characteristics and behavior of another patch, but it is meant to attract attention.

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 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. Hiding the location where an update has occurred will be present in some form with any type of patch.

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 very similar to repacking software architecture. Crane et al. apply repackaging to create beaconing ROP gadgets [75].

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 [86–91].

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. Making a system seem to have a patch at a specific location, in reality all that is implemented is a shell would be an example. Mimicking system architecture is very 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, i.e. inventing software architecture as well, or just 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 [75].

Chapter 5 addresses the application of repackaging, dazzling mimicking and decoying 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 executing. This component of deceptive patches is influenced by work studying deception within consumer advertising [92,93].

Mask Masking deploy and install chain elements of a deceptive patch involves hiding or concealing the notification text, images, sounds, etc. and identify a patch is available to install, during the installation of a patch and after the installation of a patch. Hiding this information and side channel leaks of this information 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 [94].

Repackage Repacking 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.

3.3 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 [40].

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. One limitation is 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. Another limitation is 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 to give defenders the false idea their defenses are effective. Another limitation of deceptive patches is that they do not conceal the general location of actual vulnerabilities. They can be used

to bury real vulnerable locations by injecting many false patching statements, making actual vulnerabilities more difficult to find, but this will also increase the patch size.

The main goal of deceptive patches is not to hide a vulnerability or any patched code, but to cause an attacker to mistrust any results from an exploit because he is unaware of the application's status on the victim's machine. Information leakage is something that is difficult to conceal in both traditional and deceptive patches. Deceptive patches can use information leakage advantageously by releasing false information to begin or continue a cover story.

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 [6]. 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. An attacker uses a patch to develop their exploit, so as they are studying the patch and attempting to understand its behavior and components, information gathering, deception will influence the information they gather and their subsequent actions. This analysis is important

because interrupting the cyber kill chain early effects the future steps in the sequence. Because deception impacts components within a patch that are used by the attacker to make decisions and deception is observed by an attacker, the information gathered by an attacker may be deceptive.

One interesting observation of this table is that the *weaponization* phase is not effected by deceptive patching. The attacker is not prevented from creating an exploit and in some instances, depending on the environment of the patch, s/he may be baited to make an exploit. Deceptive patching should be considered an additional defensive mechanisms such that the deception could encourage exploit development as well potentially provides defenders with additional information about adversaries and traditional defenses prevent exploits from successfully achieving their end goal, keeping the system secure.

We also note that deceptive patches impact the OODA loop [96] 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, effecting 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

	Software Architecture	System Input and Output	System Architecture	Deploy and Install Chain
Reconnaissance	Ghost Patches [7]	Active Response [70, 84, 85, 95], HoneyPatches [71]	MTD Patching	Deceptive Dispatcher
Weaponization				
Delivery	Data Patches [12, 33]			
Exploitation	Legitimate Patches	Active Response, HoneyPatches	MTD Patching	Deceptive Dispatcher
Installation			MTD Patching	
Command and Control		Active Response, HoneyPatches		
Actions on Objectives		Active Response		

Table 3.2.: Mapping Deceptive Patching tools and techniques onto the Cyber Kill Chain

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.

3.5.1 Attacker Timeline to Exploit

An attacker follows a generic timeline as they develop an exploit based on a patch. This timeline is broken into three general parts. Each part of the timeline influences the next part in that the total time to exploit is the cumulative effect of each part. We suggest there are three major time segments to consider: time to discover a patch, time to discover the vulnerability, time to develop an exploit.

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 the patch. This window is also what is referred to in related work as the time lag when an attacker develops an exploit and an end user lags behind in installing a patch on their machine. 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 is identified. This time period is what is unique to patch based exploit generation as the patch itself fixes the vulnerability, but also identifies the vulnerability being fixed.

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 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.

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. Thus, representing the exact time when an adversary downloaded a patch executable is not a key time stamp for an adversary. 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.

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

3.5.2 Timeline Analysis

Figure 3.1 represents two timelines, one with and one without deception applied. The top image clearly shows that using traditional patching techniques, exploits can be developed before machines are patched. This also shows that an exploit is developed and released during the time that benign end users are knowledgeable of a patch being available and when they actually download the patch executable. The model in the top half represents the traditional patch progression for a benign end user and for a malicious adversary developing exploits based on patches.

From the figure, a number of other metrics about attacker success and deceptive patch success can be identified. Based on the Traditional Patching Timeline, an attacker succeeds when $T_{ED} < T_{PED} + T_I$. This intuitively suggests that if an exploit can be generated at anytime before a patch executable is downloaded and installed, then an attacker's success is guaranteed. An attacker fails, which is equivalent to deceptive patch success, when $T_{ED} > T_{PED} + T_I$, $T_{VI} > T_{PED} + T_I$, or $T_{PRI} + T_{RE} > T_{PED} + T_I$. All of these cases are not explicitly modeled in the timeline, but these all suggest the same premise that increasing the time to develop an exploit past the time for end users to download and install a patch executable means success for the deceptive patch and failure for the attacker.

3.6 Deceptive Patch Formal Model Approach

A relatively new line of research considers how *deception* can be strategically employed by defenders to introduce uncertainty into an adversary's perception of

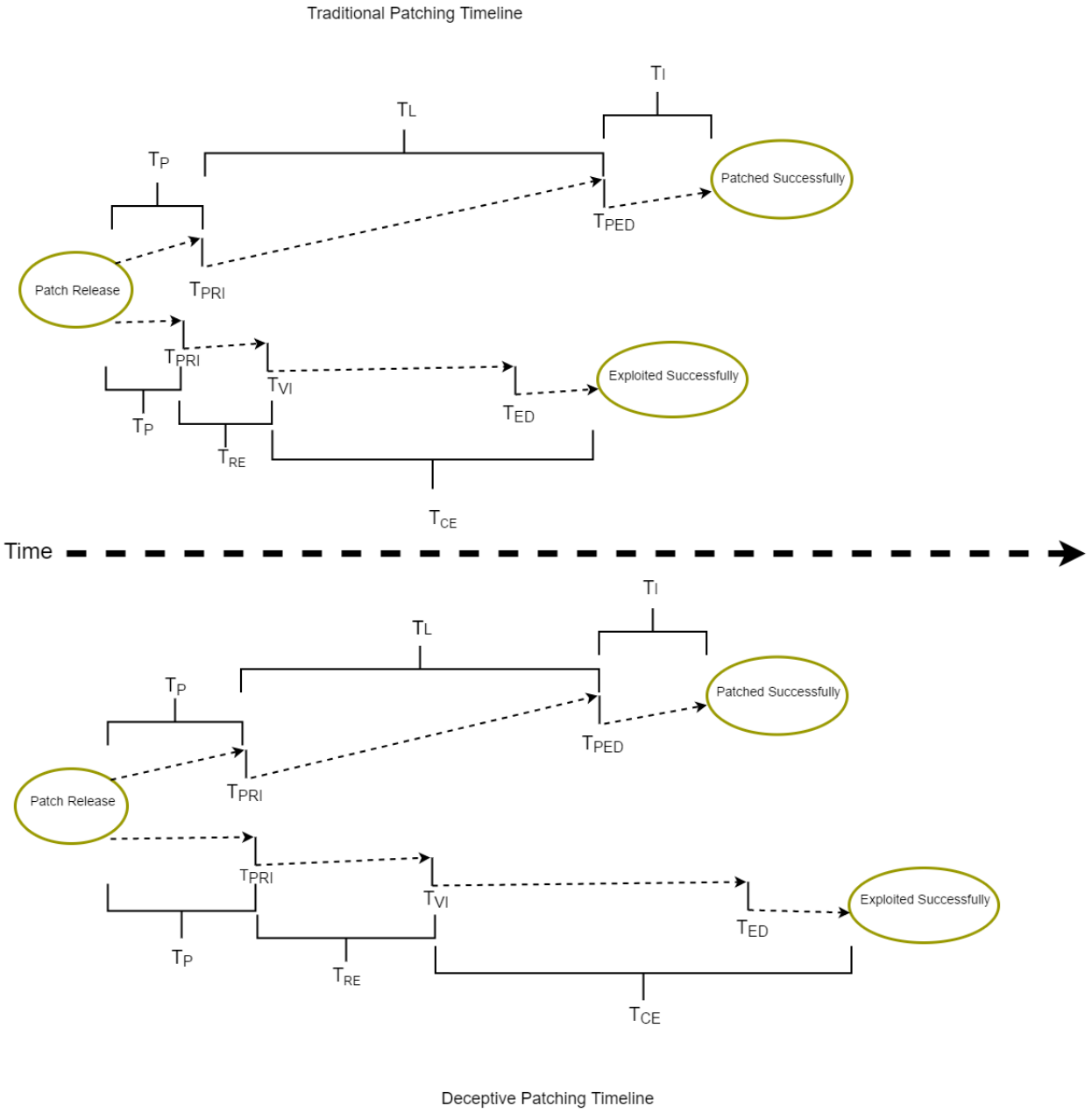

Fig. 3.1.: Difference in Traditional Patch vs Deceptive Patch Benign User and Malicious Adversary timeline

a system. Work in this area leverages results from applied security, game theory,

and cryptography to design solutions that force adversaries into a game of imperfect information.

In this work, we focus on *deceptive patches*, where standard software patches are designed to limit the knowledge an adversary can gain about the underlying vulnerability. The vulnerabilities that we consider are security flaws that directly or indirectly lead to unauthorized, non-standard, or unwarranted behavior. Patches provide direct insight into these vulnerabilities present in unpatched systems, and deceptive patching techniques have been proposed to limit this information gain. There are three general categories of deceptive patches: *faux* patches that introduce a fix for a vulnerability that does not actually exist, *obfuscated* patches that hide the vulnerability being addressed, and *active response* patches that fix the vulnerability yet attempt to convince adversaries that the system remains unpatched.

Deception is starting to gain traction as more companies and security tools are applying deceptive principles as defensive strategies. Deception has a number of disadvantages, chief among them is that deceptive data, tools, behavior is ideally difficult to distinguish from their legitimate counterparts, thus attackers waste resources or expose themselves while defenders only incur linear costs. This is a win for the defense. Though intuitively this paradigm is understood, a formal representation of why deception works is lacking. 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 can be used to identify theoretically secure techniques as well as those that fall short of theoretical security, but additional analysis shows they could still be effective in practice.

Existing results on deceptive patching techniques argue their utility informally, but lack the rigorous definition of security properties that is standard in theoretical cryptography. As an emerging research area, it is important to develop a strong

foundation from which to reason about the security of proposed techniques. As we will see, many of the defensive techniques that have been proposed cannot achieve their desired security guarantees under reasonable assumptions. Thus, it is critical that the field adopts the rigorous treatment common in theoretical cryptography, rather than rely on the flawed historical approach of security-by-obscurity arguments.

We first review prior work on deceptive patching techniques, and then introduce formal game-based security definitions that capture the technique's claimed security guarantees. Finally, we discuss whether these ideal properties of deceptive systems can be achieved in reality.

We aim to present a methodology to formally model the security of different types of deceptive patches. This treatment is a preliminary attempt to do so.

The original code is a simple string copy example where the input string has a length that is not identified. This could result in the string buffer being overwritten.

3.7 Modeling Security of Deception

The security of cryptographic protocols is typically demonstrated through either a game- or simulation-based security proof. 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 computationally indistinguishable from an ideal-world construction. In this work, we adopt the game-based approach. 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. This suggests that both *ghost* and *obfuscated* patches cannot satisfy a meaningful security definition. In contrast, *active response* patches do preserve this

Original

```
int vul_func(char *input_string,
    int input_length){
    char string[20]

    strcpy(string, input_string);

    return 0;
}
```

Faux

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

Obfuscated

```
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;
}
```

Active Response

```
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;
}
```

Table 3.3.: Deceptive Patch Examples

indistinguishability, as the response from both unpatched and patched programs is equivalent. This suggests that these patches can satisfy a meaningful security definition in practice. To analyze the security for all three types of patches, we choose to model them using a game-based security definition.

3.7.1 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 given in unary as 1^λ. In Section 3.8.2, we introduce a notion of λ in the context of deceptive patches.

3.7.2 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 in order to prevent an adversary from constructing an efficient distinguisher.

3.7.3 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 tech-

niques. However, we shall see an example of a deceptive technique where `Verify` may not allow the construction of an efficient distinguisher.

3.7.4 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 samples of both legitimate and faux patches, as both are generally publicly available. $\mathcal{A}$ also can interact with unpatched, legitimately patched and faux patched code with no time restraints.

Indistinguishability Game

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

The Faux Patch Indistinguishability game of Protocol 3.7.1 proceeds as follows: (1) Adversary $\mathcal{A}$ requests a polynomial number of patches $\mathtt{P}$ sampled from the set of legitimate patches $\mathcal{L}$. (2) The Oracle responds with random $\mathtt{P} \xleftarrow{\$} \mathcal{L}$. (3) Similarly, $\mathcal{A}$ requests a polynomial number of patches $\mathtt{P}$ sampled from the set of faux patches $\mathcal{F}$. (4) The Oracle responds with random $\mathtt{P} \xleftarrow{\$} \mathcal{F}$. (5) $\mathcal{A}$ requests a challenge patch $\mathtt{P'}$. (6) The system tosses a coin $b \in \{0, 1\}$, which determines whether the patch is sampled from $\mathcal{L}$ or $\mathcal{F}$. Thus, $\mathcal{A}$ must distinguish whether $\mathtt{P'}$ is a legitimate or faux patch. (7) Adversary $\mathcal{A}$ optionally requests a polynomial number of legitimate patches $\mathtt{P} \xleftarrow{\$} \mathcal{L}$ that have not been queried before, and such that $\mathtt{P} \neq \mathtt{P'}$. (8) The Oracle responds with sampled patches $\mathtt{P} \xleftarrow{\$} \mathcal{L}$. (9) Similarly, $\mathcal{A}$ optionally requests a

[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 [97].

Protocol 3.7.1: $\mathtt{P}-$IND Patch Indistinguishability

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

polynomial number of faux patches $\mathtt{P} \xleftarrow{\$} \mathcal{F}$ that have not been queried before, and such that $\mathtt{P} \neq \mathtt{P'}$. (10) The Oracle responds with sampled patches $\mathtt{P} \xleftarrow{\$} \mathcal{F}$. (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.

3.7.5 Obfuscated Patches

Adversarial Model

We consider an adversary $\mathcal{A}$ bound to PPT that attempts to identify the underlying vulnerability hidden within an *obfuscated* patch. We assume that $\mathcal{A}$ has access to the obfuscated patch, as they are generally publicly available. $\mathcal{A}$ also can interact with patched and unpatched software to validate whether the exploit they have identified exists for an unlimited amount of time.

Vulnerability Identification Game

In the Obfuscated Patch Identification game, an adversary $\mathcal{A}$ is asked to identify a vulnerability $\mathcal{V}$ within an obfuscated patch ($\texttt{P} \in \mathcal{O}$).

Protocol 3.7.2: $\mathcal{V}-$ID Vulnerability Identification

	Adversary $\mathcal{A}$		Patch Oracle $\mathcal{O}$	
(1)	Request $\texttt{P} \in \mathcal{O}$, $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$\texttt{P} \xleftarrow{\$} \mathcal{O}$, $0 \leq i \leq \text{poly}(\lambda)$	**(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 Challenge	$\longrightarrow$		
		$\longleftarrow$	$\texttt{P'} \xleftarrow{\$} \mathcal{O}$	**(6)**
(7)	$\texttt{Identify}(\mathcal{V} \in \texttt{P'})$			
(8)	$\texttt{Verify}(\mathcal{V} \in \texttt{P'})$			
(9)	$\mathcal{V}$	$\longrightarrow$		
			$b \leftarrow \texttt{Validate}(\mathcal{V} \in \texttt{P'})$	**(10)**

The Vulnerability Identification game of Protocol 3.7.2 proceeds as follows: (1) Adversary $\mathcal{A}$ requests a polynomial number of patches $\texttt{P}$ sampled from the set of

obfuscated patches $\mathcal{O}$. (2) The oracle responds with random $\texttt{P} \xleftarrow{\$} \mathcal{O}$. (3) $\mathcal{A}$ attempts to identify the vulnerability obfuscated in each patch P. (4) $\mathcal{A}$ attempts to validate the legitimacy of the identified vulnerability obfuscated in each patch P. (5) $\mathcal{A}$ requests a challenge patch P'. (6) The oracle responds with a patch $\texttt{P'} \xleftarrow{\$} \mathcal{O}$ which has not been returned to $\mathcal{A}$. (7) $\mathcal{A}$ attempts to identify the vulnerability $\mathcal{V}$ obfuscated in the challenge patch P'. (8) $\mathcal{A}$ attempts to verify the vulnerability identified in the obfuscated challenge patch P'. (9) $\mathcal{A}$ sends the identified vulnerability $\mathcal{V}$ to the oracle. (10) The system validates whether $\mathcal{V}$ is the obfuscated vulnerability in P' and responds with bit $b = 1$ if this is true, and otherwise returns $b = 0$.

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.

3.7.6 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 P or an unpatched state $\bar{\texttt{P}}$. 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.

Protocol 3.7.3: 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)	$c'_i \in \mathcal{C}_{\mathtt{P}}, \mathbf{c} \neq c'_i$, $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$r'_i \leftarrow \mathtt{P}(c'_i)$, $0 \leq i \leq \text{poly}(\lambda)$	**(8)**
(9)	$c'_i \in \mathcal{C}_{\bar{\mathtt{P}}}, \mathbf{c} \neq c'_i$, $0 \leq i \leq \text{poly}(\lambda)$	$\longrightarrow$		
		$\longleftarrow$	$r'_i \leftarrow \bar{\mathtt{P}}(c'_i)$, $0 \leq i \leq \text{poly}(\lambda)$	**(10)**
(11)	Guess **b'** $\stackrel{?}{=} b$	$\longrightarrow$		

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 just the content of the response, but also auxiliary information `Aux` (e.g., packet delay).

The Active Response Patch Indistinguishability game of Protocol 3.7.3 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) $\mathcal{A}$ selects a challenge $\mathbf{c}$ which has not been issued previously, and queries the system. (6) The system tosses a coin $b \in \{0,1\}$, 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 $\mathrm{Adv}_{\mathcal{A}}^{\mathtt{AR}-\mathrm{IND}} = Pr[\mathbf{b}' = b]$ represent the probability of $\mathcal{A}$ winning the game. We require that the advantage of a PPT adversary $\mathcal{A}$ is $|\mathrm{Adv}_{\mathcal{A}}^{\mathtt{AR}-\mathrm{IND}} - \frac{1}{2}| \leq \epsilon$ where ϵ is negligible in the security parameter λ.

3.8 Achieving the Ideal

In order 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, many of the claimed security properties of deceptive patching are unable to be realized under a formal security model.

3.8.1 Obfuscated Patches

The modern approach to cryptography requires formal security definitions based on the presumed difficulty of computationally bounded adversaries from solving well-studied mathematical problems. This is in contrast to the historical "security by obscurity" approach, which relied on adversaries lacking knowledge of the cipher design. 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 [98], 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 [99].

An adversary $\mathcal{A}$ with access to `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 development in polynomial time and code analysis even with the application of obfuscation [32, 100]. Because adversaries have access to the actual code, tests can be generated using these techniques to verify input as an exploit or identify the underlying vulnerability.

Because obfuscated patches alter the semantics of a program, an adversary can use the response of a program to an exploit to determine the vulnerability that is being fixed.

3.8.2 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 $|\text{Adv}_{\mathcal{A}}^{\mathtt{AR}-\text{IND}} - \frac{1}{2}| \leq \epsilon$ in Protocol 3.7.3.

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 3.9). 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 range 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 range, 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. 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

[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 3.7.3 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.

security of the deception is either directly or inversely related to the size of λ (i.e. larger the security parameter the more security the patch is and vice versa or the smaller the security parameter more secure and vice versa).

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 of the vulnerability being patched. In general, this can be thought of as 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 easy to model. On the other hand, 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 given a vulnerability and an exploit is unknown. 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.

3.9 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.

3.9.1 VM 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 [101]. 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 [74]. These VMs could also be detected by the adversary through timing analysis, and identifying the presence or absence of hardware and drivers [102]. 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 [101], 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.

3.9.2 Non-VM 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 [101]. Thus, other approaches to implementing active response patches must be explored that do not use the sandbox technique.

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 [6]. 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 [103]. 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.

3.10 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 this dissertation fills some of these gaps. We also analyze an economic model of deceptive patches and visualize how they influence patch based exploit generation. Finally, this chapter presents a first step towards formally modeling the security of proposed deceptive defense techniques. Our results show that many of these techniques are unable to satisfy a meaningful definition of theoretical security. However, we have argued that some implementations of active response patches can provide tenable security by satisfying a clear and meaningful security definition.

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 effect 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

Applying deceptive techniques to software level is the most technical of all the deceptive patching techniques. Software architecture deception also potentially has the most flexibility as there is a wide range of code that can be played with and altered and presented. Also, this level of defense is the last line of defense between attackers and the patch. Thus, defenses at this level mainly fall into preventive techniques and limiting access, but when these defenses are defeated, there is no additional level of defense. This is where deceptively altering the code within a patch can be applied to add extra layers of defense and even help defenders gain information about attackers.

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

We separate the background section into major techniques of software architecture deception.

Software developers release programs to the public every day, but this code is not perfectly written. As stated by Mosher's Law of Software Engineering: *"Don't worry if it doesn't work right. If everything did, you'd be out of a job."* [104]. Thus developers release code that has flaws and subsequently provide the necessary patches to fix the flawed code.

These patches are released with varying frequency, depending on the severity of the flaw as well as developer resources. Software security patches have a higher severity rating than non-security patches because these vulnerabilities could negatively impact other programs and services present on the machine. Thus, when a security flaw is discovered, a patch to fix this flaw usually follows shortly after. This trend is shown by a report released in 2016 where the average time the top five most frequently exploited zero-day vulnerabilities remained undetected once an attack was released was 7 days and the average time to install a patch for these vulnerabilities once the patch was released was 1 day [105].

Despite the speed with which these flaws are detected and updated, patches also inherently have a negative impact on the software: revealing the location of vulnerabilities and the type of security vulnerability being patched. This provides a blueprint for attackers as they develop exploits for vulnerable systems. Exploit generation research shows that attackers can use patches to find vulnerabilities and use these results to develop exploits for unpatched code [38, 61].

One solution to make this blueprint more difficult to understand is to obfuscate the traditional patch [33, 68, 79]. This approach will make the patch more difficult to statically analyze, but dynamic analysis tools exist that can analyze obfuscated code [32] and the location of the patch is not masked. Therefore, additional techniques

must be applied to software patching to enhance its security. This chapter specifically explores using deception to enhance the security of input validation patches.

4.2 Technical and Approach Background

Deception Our research will use decoy document properties to develop fake patches. The definition of deception in computation states that any intentional act of misleading to influence decision making classifies an act as deception [54, 55]. Thus, *ghost patches* are intentionally placed in code to mislead attackers. Deception can also be broken into two components, simulation, showing the false, and dissimulation, hiding the real. Simulation and dissimulation are broken down into three separate components. Simulation is comprised of mimicking, inventing, and decoying and dissimulation is comprised of masking, repackaging, and dazzling [57].

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 [106]." 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 [37,38]. 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 [107]. 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(es) 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 patch 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. 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 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. Due to performance constraints, embedded or real time systems do not present a suitable environment for ghost patches.

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

4.4 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 [45, 64] and created a list of properties that decoy documents should embody. We slightly modify 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.

4.5 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.

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

Table 4.1.: Fake Patch Properties

Mimicking this trend could deceive attackers by showing changes that are expected but fake, addressing the *enticing* property. Experimentation will show how fake patches effect 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 address 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.

4.6 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 dependant 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.

4.7 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*. 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 to the traditionally patched file. The fake patches are also implemented in intermediate representation bytecode. 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.

4.8 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

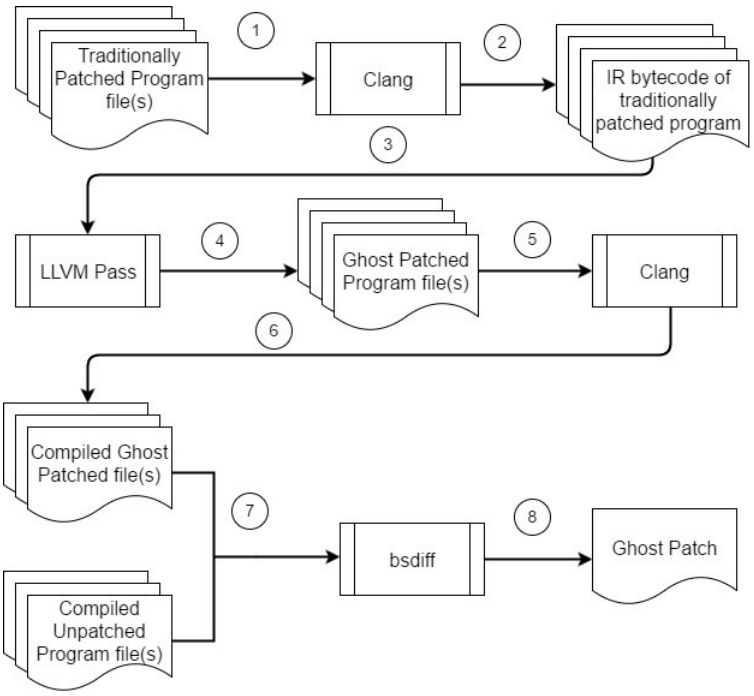

Fig. 4.1.: Complete flow to create a Ghost Patch using LLVM and bsdiff

types and data structures without loss of generality. Our implementation uses LLVM and is about 900 lines of C++ code.

The prototype of the faux patch program was developed using an LLVM pass on an Ubuntu 14.04 x86_64 virtual machine. We used LLVM (version 3.4) to develop our pass because its front end compiler allows optimizations to be developed that can be applied to programs agnostic of the language they are written in. We use KLEE [107] (version 1.3) to dynamically analyze our vulnerable code because it evaluates paths through a program using symbolic execution, which can efficiently analyze programs without enumerating every possible input value. Thus, results from Klee represent a best case scenario for attacker resource utilization. The VM has 2 cores and 4GB RAM. All experiments were also run on this virtual machine.

4.8.1 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].

Experimentation To evaluate our approach, we compare the length of time for Klee [107], 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 measure the impact of a faux patch on exploit generation. We exploit 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 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 fake patch's effect on program performance.

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

4.9 Results

4.9.1 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.

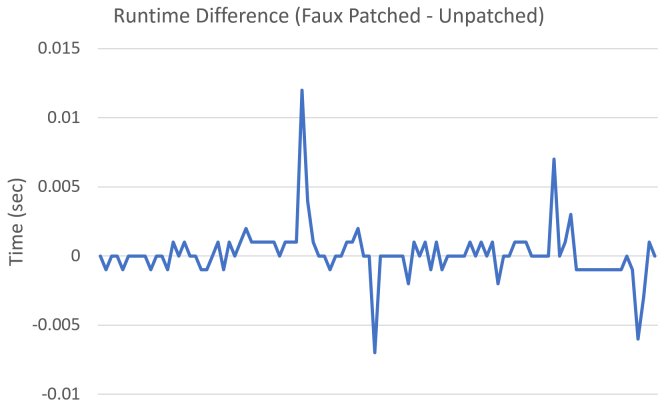

Fig. 4.2.: Difference in Faux Patched vs. Unpatched program runtime

4.9.2 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 each program across 100 executions. A t test using these values reveled 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 traditionally patched program, thus analyzing ghost patches would also require more resources.

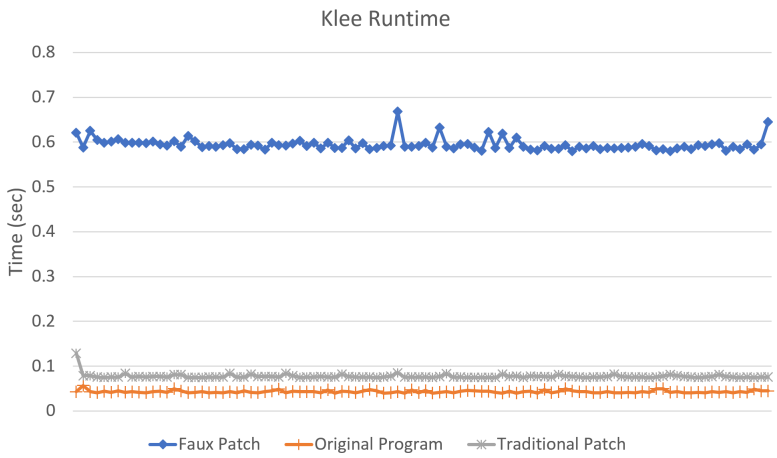

Fig. 4.3.: Klee runtime analysis for simple program

Part of the analysis for faux patches uses programs from the klee-benchmark as input. As part of this analysis, three programs performed as expected (i.e. increasing the average number of paths for Klee to analyze). Figure 4.4 represents the runtime for each of the three programs across 20 runs for both a faux patched version of the program and an unpatched version of the code.

4.10 Discussion

Based on testing of the Klee-benchmarking suite, this section provides a more depth analysis of how ghost patching impacts these programs. Of the 15 programs tested, the average Klee analysis runtime for analyzing faux patched programs increased 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 due to increasing the number of paths to analyze. This is supported by our

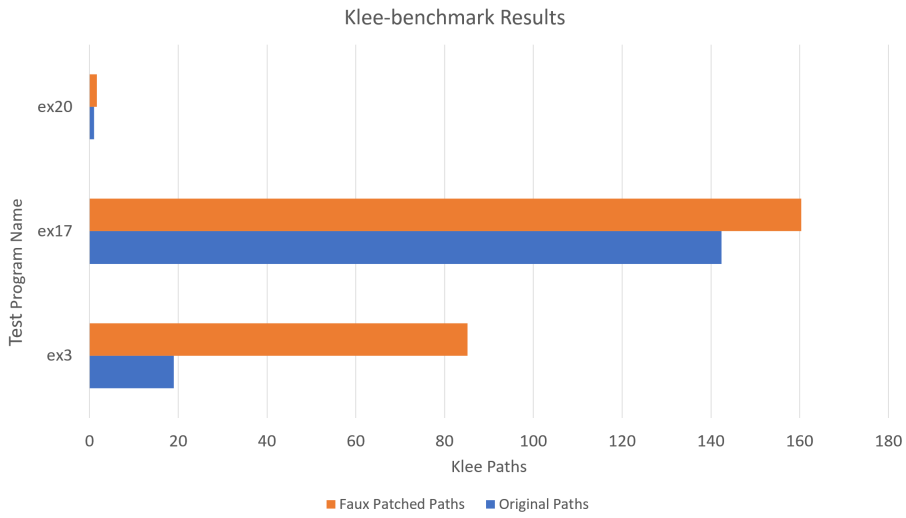

Fig. 4.4.: Klee runtime analysis for klee-benchmark programs

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 of Klee changes, 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 in 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, increases 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 randomly selected value used in the conditional statements we well as the comparison operator used in these statements.

Patch Obfuscation There are limitations associated with ghost patches that could provide attackers an advantage in identifying fake patches or minimizing their impact on program analysis. 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 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, OS 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 will 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 change 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 [71]. This masking would increase the resources necessary for dynamic analysis tools to identify unpatched systems. Further research will develop techniques that hinder or prevent exploit verification. An overview of active response patches is provided in Chapter 3.

Approach Limitation Another limitation is that based on our experiments, 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 that 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 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.

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. We believe that with additional research and testing, this approach, either as a standalone 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 above supports this claim.

4.11 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. The work on ghost patching appears in IFIP SEC 2017 [7].

5. DECEPTIVE DISPATCHER: COMBINING DECEPTIVE SYSTEM ARCHITECTURE AND 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, a 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 framework is also described that adds deception to the current software patch lifecycyle by combining deceptive system architecture and 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 to 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

5.1 Example Application of MTD to Software Security Patches

MTD can be applied to software security patches in a variety of ways. 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 framework for created 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* and *(*GetWmicObject). 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 once a patch is released where notifications occur. The first identifies that a patch is available to download. This notification can be presented through an updating 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. Information such as files that are being added, edited and/or removed, progress of the installation, restarting the computer, and prompts for users to accept or decline actions the computer is performing. 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 is the sociology and psychological fields, but little has been done in the security field. 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 it just meant to give end users a sense of work being done [48].

Deceptive Text and Bias Prior work by Pfleeger et al. has studied behavioral science and its impact on cyber security tool development [110]. Ding et al. research how to create a dictionary of words from phishing emails that elicit biases [111].

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. A formal treatment of the decision-making process is called the OODA loop [96]. 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 will 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 [112]. Almeshekah et. al provide an overview of bias and its role in deception [113].

Using deceptive patches exploits 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. [71, 72] supports using deceptive patches to improve system defense.

Deceptive Semantic Generation Because notifications and alters are written in text and statically presented, using advertising techniques to deceive users is a viable application. 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 spotlight for attackers to investigate the associated program for vulnerabilities and develop a corresponding exploit. Research and enterprise defenses focus around the speed and efficiency of patching systems while they are in use [11, 114, 115]. Research on adding deception into the current software security protocol is scarce. Both software diversity as a MTD deceptive technique and deploy and install chain deception can be combined to generate and release deceptive patches using current technologies that cast uncertainty on this spotlight. 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 framework suggests releasing diversified or 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 but with the same output as the original patch. The main point of this approach though is that the re-released patch addresses the same vulnerability but just 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 diversi-

fied 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:** [36]
   (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 patch 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 patch that was addressed by P_O, have "enough diversity" when compared to P_O, and it is 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 could 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 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. This protocol also only duplicates the existing security in place by the original patch release P_O. Subsequent patches do not remove security from the program. We explain this visually represent the the protocol in Figure 5.1.

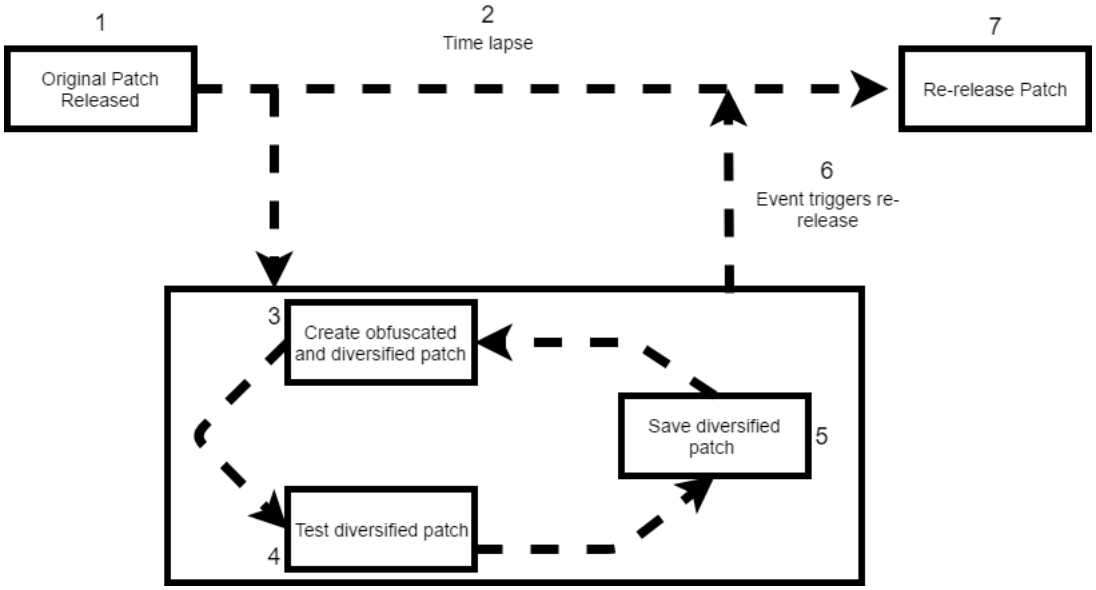

Fig. 5.1.: High level overview of re-release protocol

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 3 provides more detail 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. Thus, P_n can be thought of as a new band-aid for a scar that had an old band-aid. The new band-aid appears and performs differently, but execute the same functionality as the original. This approach forces an attacker to identify whether a released patch is an original patch that fixes an original vulnerability that is present or if is a diversified re-release that still address the original vulnerability, but because end users have ideally applied the original patch, 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 an unintended 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 send 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.

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 a certain 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 could 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.

Candidate Re-release Patches Deciding which types of patches can be diversified is important for developers to efficiently create re-release patches. One metric that can be used is the length of a patch. The number of lines of code could provide more code to diversify. Another metric to use is to diversify the patch and compare versions to measure the diversity between the versions [60, 86, 116]. If the diversity measure is above some threshold, then the diversified patch can be saved and re-released at a later date.

Discussion: Side Effects and Limitations From an end users perspective, side effects are positive. As end users, their code ideally 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 change to protect their systems, something that does not happen in the current patching ecosystem. The question to ask with this approach is that if an end user does not apply the original patch, what probability is there that they 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, though is outside of the scope of this dissertation.

From an attackers 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 wastes 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 patch their systems, the potential for them to apply re-released patches is also grave. Developers cannot force end users to apply a patch for their software. Future work will research identify methods that can increase the probability of patch installation.

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 a gives general overview of biases that are influenced based on the presence of deception. A framework is also described that adds deception to the current software patch life-cycle by combining deceptive system architecture and 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 to 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. 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 discuss the literature providing background material for deceptive patches as well as prior work in the area of deceptive patching. We then identify and discuss the major components of a software patch in Chapter 3. We discuss how these components can be deceptively implemented as well as how adding deception to each components impacts the cyber kill chain. We present a timeline analysis of the effect of deceptive patches and finally analyze a formal model of deceptive patches that examines the theoretical security of deceptive patches using a game theoretic approach.

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 discuss implications of this work.

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 and add subsequent steps to generate diverse versions of released patches and discuss metrics that trigger the release of these diversified patches.

6.1 Future Work

The research conducted and presented in this dissertation provide 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 and can be used as a stepping stone to conduct future work.

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 shed light on additional characteristics that must be present that 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 huge impact on the way we perform defense. Classification and clustering techniques can be used to identify and develop the most 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

is one step 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 tool tool is future work.

REFERENCES

REFERENCES

[1] Symantec, "Internet security threat report," Symantec, Tech. Rep., 2015.

[2] V. Enterprise and Affiliates, "2015 data breach investigations report," Verizon Enterprise Solutions, Tech. Rep., 2015.

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

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

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

[6] 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.

[7] J. Avery and E. H. Spafford, "Ghost patches: Fake patches for fake vulnerabilities," in *32nd International Conference on ICT Systems Security and Privacy Protection*. IEEE, 2017.

[8] J. Avery, M. Almeshekah, and E. Spafford, "Offensive deception in computing," in *12th International Conference on Cyber Warfare and Security 2017 Proceedings*, 2017, p. 23.

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

[10] "Patch (computing)," 2016. [Online]. Available: 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 *Eleventh Annual 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 *IEEE Symposium on Security and Privacy*. IEEE, 2007, pp. 252–266.

[13] D. Zamboni, "Using internal sensors for computer intrusion detection," *Center for Education and Research in Information Assurance and Security*, 2001.

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

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

[18] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack, "Timing the application of security patches for optimal uptime." in *LISA*, 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," *ICIS 2006 Proceedings*, p. 22, 2006.

[22] J.-W. Sohn and J. Ryoo, "Securing web applications with better "patches": An architectural approach for systematic input validation with security patterns," in *10th International Conference on Availability, Reliability and Security*. IEEE, 2015, pp. 486–492.

[23] 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 *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 234–242.

[24] 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.

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

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

[27] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267349

[28] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*. ACM, 2006, pp. 131–138.

[29] S. Rugaber, T. Shikano, and R. K. Stirewalt, "Adequate reverse engineering," in *Proceedings. 16th Annual International Conference on Automated Software Engineering*. IEEE, 2001, pp. 232–241.

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

[31] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Proceedings. Ninth working conference on Reverse engineering*. IEEE, 2002, pp. 45–54.

[32] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *12th Working Conference on Reverse Engineering*. IEEE, 2005, pp. 10–pp.

[33] C. Wang and S. Suo, *The practical defending of malicious reverse engineering*. University of Gothenburg, 2015.

[34] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*. ACM, 2006, pp. 131–138.

[35] T. Rains, "When vulnerabilities are exploited: the timing of first known exploits for remote code execution vulnerabilities," *Microsoft Secure Blog*, 2014.

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

[37] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation." in *NDSS*, vol. 11, 2011, pp. 59–66.

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

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

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

[41] A. Dewdey, *Computer Recreations, A Core War Bestiary of Virus, Worms and other Threats to Computer Memories*. Scientific America, 1985, vol. 252.

[42] K. D. Mitnick and W. L. Simon, *The art of deception: Controlling the human element of security*. John Wiley & Sons, 2011.

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

[44] C. Stoll, *The cuckoo's egg: tracking a spy through the maze of computer espionage*. Simon and Schuster, 2005.

[45] 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.

[46] J. Yuill, M. Zappe, D. Denning, and F. Feer, "Honeyfiles: deceptive files for intrusion detection," in *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop*. IEEE, 2004, pp. 116–122.

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

[48] E. Adar, D. S. Tan, and J. Teevan, "Benevolent deception in human computer interaction," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '13. New York, NY, USA: ACM, 2013, pp. 1863–1872. [Online]. Available: http://doi.acm.org/10.1145/2470654.2466246

[49] 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.

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

[51] U. S. J. C. of Staff, *JP 3-13.4: Military Deception*, ser. Joint pub. Joint Chiefs of Staff, 2006. [Online]. Available: https://books.google.com/books?id=P5l2rgEACAAJ

[52] J. Pub, "Pub 3-58: Joint doctrine for military deception," 1996.

[53] 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.

[54] J. J. Yuill, *Defensive computer-security deception operations: processes, principles and techniques*. ProQuest, 2006.

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

[56] J. Bell and B. Whaley, *Cheating and Deception*. Transaction Publ., 1991. [Online]. Available: http://books.google.com/books?id=ojmwSoW8g7IC

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

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

[59] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

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

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

[62] 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.

[63] M. Franz, "E unibus pluram: massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 workshop on New security paradigms*. ACM, 2010, pp. 7–16.

[64] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo, *Baiting inside attackers using decoy documents*. Springer, 2009.

[65] B. Whitham, "Canary files: Generating fake files to detect critical data loss from complex computer networks," in *The Second International Conference on Cyber Security, Cyber Peacefare and Digital Forensic (CyberSec2013)*. The Society of Digital Information and Wireless Communication, 2013, pp. 170–179.

[66] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 145–160.

[67] 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.

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

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

[70] 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.

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

[72] F. Araujo, M. Shapouri, S. Pandey, and K. Hamlen, "Experiences with honey-patching in active cyber security education," in *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*, 2015.

[73] T. Holz and F. Raynal, "Detecting honeypots and other suspicious environments," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*. IEEE, 2005, pp. 29–36.

[74] 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.

[75] S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Booby trapping software," in *Proceedings of the 2013 workshop on New Security Paradigms workshop*. ACM, 2013, pp. 95–106.

[76] J. Heusser and P. Malacaria, "Quantifying information leaks in software," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 261–269.

[77] 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 *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006, pp. 37–46.

[78] 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.

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

[80] S. Banescu, M. Ochoa, and A. Pretschner, "A framework for measuring software obfuscation resilience against automated attacks," in *IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 2015, pp. 45–51.

[81] 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.

[82] Y. Kanzaki, A. Monden, and C. Collberg, "Code artificiality: a metric for the code stealth based on an n-gram model," in *Proceedings of the 1st International Workshop on Software Protection*. IEEE Press, 2015, pp. 31–37.

[83] 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.

[84] H. C. Goh, "Intrusion deception in defense of computer systems," DTIC Document, Tech. Rep., 2007.

[85] 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.

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

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

[88] 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.

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

[90] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: transparent moving target defense using software defined networking," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 127–132.

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

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

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

[94] A. Arora, R. Krishnan, R. Telang, and Y. Yang, "An empirical analysis of software vendors' patching behavior: Impact of vulnerability disclosure," *ICIS 2006 Proceedings*, p. 22, 2006.

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

[96] J. R. Boyd, "The essence of winning and losing," *Unpublished lecture notes*, vol. 12, no. 23, pp. 123–125, 1996.

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

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

[99] W. Michiels, P. Gorissen, and H. Hollmann, "Cryptanalysis of white-box implementations," Report 2008/105 in Cryptology ePrint Archive, 2008, http://eprint.iacr.org/2008/105.

[100] S. Banescu, M. Ochoa, and A. Pretschner, "A framework for measuring software obfuscation resilience against automated attacks," in *IEEE/ACM 1st International Workshop on Software Protection (SPRO)*. IEEE, 2015, pp. 45–51.

[101] 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.

[102] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: Vmm detection myths and realities." in *HotOS*, 2007.

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

[104] M. L. of Engineering, *Top 50 Programming Quotes of All Time.* Tech-Source, 2010. [Online]. Available: http://www.junauza.com/2010/12/top-50-programming-quotes-of-all-time.html

[105] Symantec, "Internet security threat report," Symantec, Tech. Rep., 2016.

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

[107] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[108] S. Dolan, "mov is turing-complete," Tech. rep. 2013 (cit. on p. 153), Tech. Rep., 2013.

[109] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-third annual Applied Computer security applications conference, ACSAC.* IEEE, 2007, pp. 421–430.

[110] 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.

[111] 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 *IEEE International Conference on Semantic Computing.* IEEE, 2015, pp. 252–259.

[112] M. Bennett and E. Waltz, *Counterdeception principles and applications for national security.* Artech House Norwood, MA, 2007.

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

[114] I. Lee, "Dymos: a dynamic modification system," Ph.D. dissertation, University of Wisconsin, Madison, 1983.

[115] 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," Aug. 24 2010, uS Patent 7,784,044. [Online]. Available: https://www.google.com/patents/US7784044

[116] Y. Kanzaki, A. Monden, and C. Collberg, "Code artificiality: a metric for the code stealth based on an n-gram model," in *Proceedings of the 1st International Workshop on Software Protection.* IEEE Press, 2015, pp. 31–37.

VITA

VITA

Jeffrey K. Avery was born the youngest 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 in 2008. Afterward, he graduated *magna cum laude* from the University of Maryland, Baltimore County in 2012 with his Bachelor's degree in Computer Science where he was a Meyerhoff Scholar. That same year, he was admitted into the Computer Science PhD program at Purdue University. He received his Master's Degree, also in Computer Science, in 2014.

Jeffrey has completed a number of successful internships, including stints at Army Research Laboratory, John's Hopkins Applied Physics Laboratory, McAfee, Inc. and currently is a cyber security intern at Sypris Electronics, Inc. His research interests include network security, anti-phishing and deception. Upon graduation, Jeffrey has accepted a full-time offer at Northrop Grumman Corporation as a Software Engineer in the Future Technical Leaders program.