ENGINEERING CYBER-DECEPTIVE SOFTWARE

Frederico Araujo, B.S., M.S.

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

SOFTWARE ENGINEERING

The University of Texas at Dallas

August 2016

To my wife, Rubia.

ACKNOWLEDGMENTS

I would like to express profound gratitude to my advisor, Kevin Hamlen, for his unyielding support, unfailing patience, and invaluable guidance in the fine art of technical writing and computer science research. His knowledge, unassuming teachings, and insightful direction were pivotal in the completion of this dissertation. Moreover, his humble personality and technical breadth reassured and inspired me to achieve more than I could have ever imagined.

I also wish to thank my dissertation committee members, Latifur Khan, Zhiqiang Lin, and Bhavani Thuraisingham, for their support and encouragement throughout this process, and Rhonda Walls, for her inestimable support with conference travel arrangements, proposal and defense logistics, and for always being available when I needed her help. To all professors who have inspired me, especially Paulo Barreto, Edson Fregni, and Selma Melnikoff.

I extend my heartfelt thanks to my friend Owolabi Legunsen, who always offered me encouragement and shared wise words of advice. To my friends, Mohammad Zinatti, Meera Shridar, Vishwath Mohan, Safwan Khan, Wenhao Wang, Benjamin Ferrell, Gbadebo Ayoade, Khaled Naami, and all who made this a memorable journey.

I am most thankful to my mother, siblings, grandparents, and sisters- and parents-in-law for their unwavering love, prayers, and support. I especially thank my mother for having raised her son with a love of learning and for having offered me a good education.

Last and most significantly, I wish to express my love and appreciation for my wife, Rubia, whose unconditional love, cheerful personality, and tireless support have always lifted me up and filled my life with joy, and for God, from whom all wisdom and knowledge flow.

PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation conforms to the dissertation requirements prescribed by the "Guide for the Preparation of Master's Theses and Doctoral Dissertations at The University of Texas at Dallas," except that its formatting has been restyled to the preferences of the author to a more aesthetically pleasing book format inspired by André Miede's *Classic Thesis* style for the LaTeX document publishing system. It includes a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) are provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

Where noted, some chapters reproduce material already published or submitted for publication. In such cases, connecting texts which provide logical bridges between different manuscripts are supplied. Where the student is not the sole author of a manuscript, an explicit statement describing the student's contribution to the work and acknowledging the contribution of the other author(s) is included. The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

This research was supported in part by AFOSR award FA9550-14-1-0173, NSF awards #1054629 and #1027520, ONR award N00014-14-1-0030, and NSA award H98230-15-1-0271. Any opinions, recommendations, or conclusions expressed are those of the author and not necessarily of the AFOSR, NSF, ONR, or NSA.

ABSTRACT

Frederico Araujo, PhD

Supervising Professor: *Kevin W. Hamlen*

Committee Members: *Latifur Khan, Zhiqiang Lin, and Bhavani Thuraisingham*

Language-based software cyber deception leverages the science of compiler and programming language theory to equip software products with deceptive capabilities that misdirect and disinform attackers. This dissertation proposes language-based software cyber deception as a new discipline of study, and introduces five representative technologies in this domain: (1) honey-patching, (2) process image secret redaction, (3) deception-enhanced intrusion detection, (4) Deception as a Service (DaaS) in the cloud, and (5) moving target deception.

Honey-patches fix software security vulnerabilities in a way that makes failed attacks appear successful to attackers, impeding them from discerning which probed systems are genuinely vulnerable and which are actually traps. The traps deceive, waylay, disinform, and monitor adversarial activities, warning defenders before attackers find exploitable victims. *Process image secret redaction* erases or replaces security-relevant data on-demand from running program processes in response to cyber intrusions. This results in deceptive programs that appear operational once penetrated, but with false secrets that misdirect intruders who hijack or reverse-engineer the victim process. *Deception-enhanced intrusion detection* joins honey-patching with the science of data mining-based intrusion detection to create more effective, adaptive threat monitors that coordinate multiple levels of the software stack to catch adversaries. *Deception as a Service* leverages the massive replication and virtualization capabilities of modern cloud computing architectures to create a “hall of mirrors” that attackers must navigate in order to distinguish valuable targets from traps. Finally, *moving target deception* employs software version emulation to more effectively lure adversaries and model evolving threat and vulnerability landscapes.

A language-based approach to the design and implementation of each of these new technologies is presented and evaluated. Experiments indicate that software cyber deception can be effectively realized for large, production-level software networks and architectures—often with minimal developmental effort and performance overheads. Language-based cyber deception is therefore concluded to be a low-cost, high-reward, yet heretofore largely unexplored methodology for raising attacker risk and uncertainty, toward leveling the longstanding asymmetry between attackers and defenders in cyber warfare battlefields.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

Throughout the history of warfare, obfuscation and deception have been widely recognized as important tools for leveling the ubiquitous asymmetry between offensive and defensive combatants. In the modern era of cyber warfare, this asymmetry has perhaps never been more extreme. Despite a meteoric rise in worldwide spending on conventional cyber defensive technologies and personnel, the success rate and financial damage resulting from cyber attacks against software systems has escalated even faster, rising dramatically over the past few decades. The challenges can be traced in part to the inherently uneven terrain of the cyberspace battlefield—which typically favors attackers, who can wreak havoc by finding a single weakness, whereas defenders face the difficult task of protecting all possible vulnerabilities.

This chapter contains material previously published as: F. Araujo and K.W. Hamlen. "Embedded Honeypotting." In Cyber Deception: Building the Scientific Foundation, S. Jajodia, V.S. Subrahmanian, V. Swarup, and C. Wang (eds.), Springer, 2016. Reproduced with permission of Springer International Publishing in the Thesis/ Dissertation format via Copyright Clearance Center. Lead author Araujo conducted the majority of the research and co-wrote the article.

The attack surface exposed by the convergence of computing and networking poses particularly severe asymmetry challenges for defenders. Our computing systems are constantly under attack, yet the task of the adversary is greatly facilitated by information disclosed by the very defenses that respond to those attacks. This is because software and protocols have traditionally been conceived to provide informative feedback for error detection and correction, not to conceal the causes of faults. Many traditional security remediations therefore advertise themselves and their interventions in response to attempted intrusions, allowing attackers to easily map victim networks and diagnose system vulnerabilities. This enhances the chances of successful exploits and increases the attacker's confidence in stolen secrets or expected sabotage resulting from attacks.

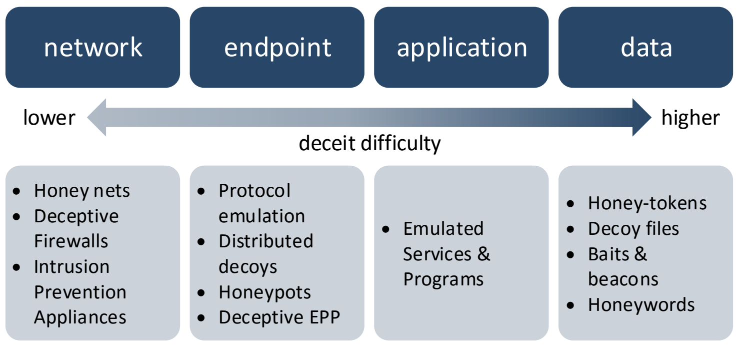


Figure 1.1: Gartner's *deception stack*, with examples of inhabiting deceptive technologies

Deceptive capabilities of software security defenses are an increasingly important, yet underutilized means to level such asymmetry. These capabilities mislead adversaries and degrade the effectiveness of their tactics, making successful exploits more difficult, time consuming, and cost prohibitive. Moreover, deceptive defense mechanisms entice attackers to make fictitious progress towards their malicious goals, gleaning important threat information all the while, without aborting the interaction as soon as an intrusion attempt is detected. This equips defenders with the ability to lure attackers into disclosing their actual intent, monitor their actions, and perform counterreconnaissance for attack attribution and threat intelligence gathering.[1]

[1] *Araujo and Hamlen (2016)*

1.1 THE DECEPTION STACK

Deceptive techniques can be introduced at different layers of the software stack. Figure 1.1 illustrates this *deception stack*,[2] which itemizes various deceptive capabilities available at the network, endpoint, application, and data layers. For example, computer networks known as *honeynets*[3] intentionally purvey vulnerabilities that invite, detect, and monitor attackers; endpoint protection platforms[4] deceive malicious software by emulating diverse execution environments and creating fake processes at the application level to manipulate malware

[2] *Pingree (2015)*

[3] *Spitzner (2003a); Thonnard and Dacier (2008)*

[4] *Pingree (2015)*

behavior; and falsified data can be strategically planted in decoy file systems to disinform and misdirect attackers away from high-value targets.[1]

[1] *Yuill et al. (2004); Voris et al. (2015)*

The deception stack suggests that the difficulty of deceiving an advanced adversary increases as deceptions move up the stack. This is accurate if we compare, for instance, the work associated with emulating a network protocol with the challenge of crafting fake data that appear legitimate to the attacker: Protocols have clear and precise specifications and are therefore relatively easy to emulate, whereas there are many complex human factors influencing whether a specific datum is plausible and believable to a particular adversary. In general, deceptive software defenses must employ one or more forms of deception, and leverage all layers of the deception stack to some degree in order to be effective against a persistent and skilled adversary.

THESIS STATEMENT: This dissertation draws attention to the disfavorable predictability of today's software, and responds by proposing and investigating scientific methodologies for engineering application-level, cyber-deceptive software, toward increasing attacker risk and leveling attacker-defender asymmetry. Such methodologies are argued to be low-cost, high-reward (yet heretofore underutilized) approaches for improving the security of large networks and software systems difficult to secure by more conventional means.

Application-level deceptive capabilities, which are the focus of this dissertation, are particularly under-researched, yet offer critical mediation capabilities between the network, endpoint and data deception layers of the deception stack. For example, an application-level, deception-enabled web server can ask the network-level firewall to allow certain payloads to reach the application layer, where it can then offer deceptive responses that misdirect the adversary into attacking decoy machines within the endpoint layer. These decoys can appeal to the data deception layer to purvey disinformation in the

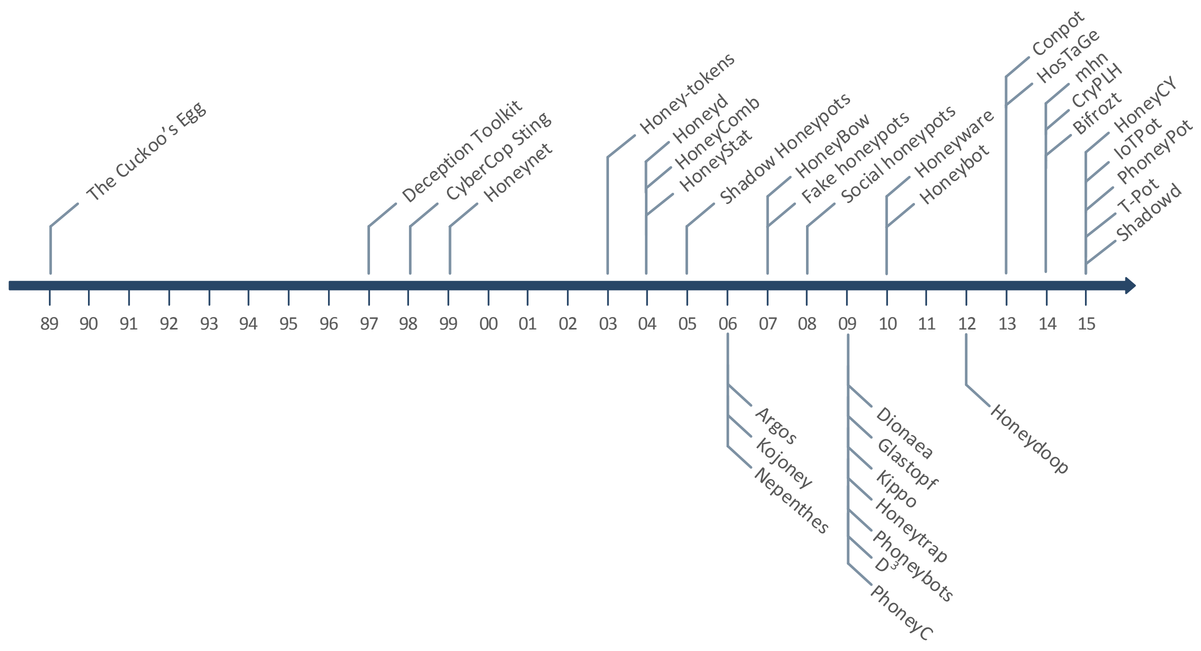


Figure 1.2: Timeline of selected academic and industrial research on honeypots

form of false secrets or even malware counter-attacks against adversaries. Such scenarios demonstrate the ongoing need for tools and techniques allowing organizations to engineer applications with proactive and deceptive capabilities that degrade attackers' methods and disrupt their reconnaissance efforts.

1.2 TRADITIONAL HONEYPOTS

The deceptive software innovations here introduced differ foundationally from traditional honeypot technologies. *Honeypots* are information systems resources conceived purely to attract, detect, and gather attack information.[1] Figure 1.2 presents an abbreviated timeline of the extensive history of honeypot research. (See Bringer et al.[2] for a more comprehensive survey on recent advances and future trends in honeypot research.) In contrast, the application-level deceptions advanced in this dissertation seek to integrate deceptive

[1] *Spitzner (2002)*

[2] *Bringer et al. (2012)*

capabilities into information systems with genuine production value (e.g., servers and software that offer genuine services to legitimate users).

Traditional honeypots are usually classified according to the interaction level provided to potential attackers. *Low-interaction* honeypots[1] present a façade of emulated services without full server functionality, with the intent of detecting unauthorized activity via easily deployed pseudo-services. *High-interaction* honeypots[2] provide a relatively complete system with which attackers can interact, and are designed to capture detailed information on attacks. Despite their popularity, both low- and high-interaction honeypots are often detectable by informed adversaries (e.g., due to the limited services they purvey, or because they exhibit traffic patterns and data substantially different than genuine services). Application-level deceptions potentially overcome this longstanding deficiency of traditional honeypots by offering much higher interactivity, facilitated by their purveyance of genuine services and their access to genuine data.

[1] *Provos (2004); Kippo (2009); Glastopf (2009); ThreatStream (2014); Deutsche Telekom AG (2015)*

[2] *Bifrozt (2014); Shadowd (2015)*

1.3 APPLICATION-LEVEL SOFTWARE DECEPTION

As a flagship example of application-level deceptiveness, honey-patching (presented in Chapter 2) introduces subterfuge into the ubiquitous regimen of security patching applied to production software systems. Patching continues to be the most pervasive and widely accepted means for addressing newly discovered security vulnerabilities in commodity software products. In 2014 alone, major software vendors reported over 7000 patched (or soon-to-be-patched) separate security vulnerabilities to the National Vulnerability Database, almost 25% of which were ranked highest severity.[3] However, despite the increasingly prompt availability of security patches, a majority of attacks in the wild continue to exploit vulnerabilities that are known and for which a patch exists.[4] This is in part because patch adoption is not immediate, and may be

[3] *Florian (2015)*

[4] *Bilge and Dumitras (2012); Fritz et al. (2013); Arbaugh et al. (2000)*

slowed by various considerations, such as patch compatibility testing, in some sectors.

As a result, determined, resourceful attackers often probe and exploit unpatched, patchable vulnerabilities in their victims. For example, a 2013 security audit of the U.S. Department of Energy revealed that 60% of DoE desktops lacked critical patch updates, leading to a compromise and exfiltration of private information on over 100,000 individuals.[1] The prevalence of unpatched systems has driven the proliferation of tools and technologies via which attackers quickly derive unique, previously unseen exploits from patches,[2] allowing them to infiltrate vulnerable systems.

[1] *Friedman (2013)*

[2] *Brumley et al. (2008)*

Attackers are too often successful at finding and exploiting patching lapses because conventional software security patches advertise rather than conceal such lapses. For example, a request that reliably yields garbage output from an unpatched server, but that reliably yields an error message from a patched server, readily divulges whether each server is vulnerable. Cyber-criminals therefore quickly and efficiently probe large networks for vulnerable software, allowing them to focus their attacks on susceptible targets.

To combat this, honey-patches take the alternative approach of patching software security vulnerabilities in such a way that future attempted exploits of the patched vulnerabilities appear successful to attackers. This masks patching lapses, impeding attackers from easily discerning which systems are genuinely vulnerable and which are actually patched systems masquerading as unpatched systems. Honey-patches offer equivalent security to conventional patches, but respond to attempted exploits by transparently redirecting the attacker's connection to a carefully isolated *decoy* environment, which monitors and disinforms criminals.

Honey-patches therefore cope with defender risk (e.g., the risk of patching lapses) by reflecting some of that risk to adversaries. By making software security patches invisible to attackers, all assets appear equally vulnerable—but many of the apparent vulnerabilities are actually traps designed to waylay and mislead. Such deception is useful in contexts where immediate, comprehensive, and universal patch deployment—the conventional defense—is infeasible or impractical. For computer networks that are large and complex, such infeasibility is common. For example, the U.S. Department of Defense has historically struggled to even track an accurate inventory of its many digital assets,[1] much less ensure that all its systems remain fully patched. In such contexts, imbuing the inventoried assets with deceptive capabilities can implicitly help to protect the non-inventoried assets.

[1] *DoD Comptroller (2015)*

Honey-patches arm live, commodity server software with deceptive attack-response and disinformation capabilities. They therefore differ from traditional honeypots in that the deceptive capabilities reside within the actual, mission-critical software systems that attackers are seeking to penetrate; they are not independent decoy systems. This affords defenders advanced deceptive remediations against informed adversaries who can identify and avoid traditional honeypots. These new capabilities mislead advanced adversaries into wasting time and resources on phantom vulnerabilities and decoy file systems, and potentially turn every attack into an information-gathering opportunity for defenders.

Because they reside within systems offering genuine services, honey-patching design and implementation raises significant challenges not faced by traditional honey-systems. For example, the introduction of deceptive capabilities must not degrade the performance or intended functionality of the production systems they augment. A principled approach to the design of such deceptive capabilities is therefore needed, drawing upon scientific methodologies related to software architecture, compiler design and implementation, and program analysis.

The approach therefore constitutes a new language-based security methodology,[2] and paves the way for an emerging science of *deception-facilitating software engineering*. In particular, Chapter 3 details a new compiler technology for dynamic information flow tracking that aids in deceptive software engineering, Chapter 4 investigates deception-powered

[2] *Schneider et al. (2001)*

enhancements to anomaly-based intrusion detection architectures, Chapter 5 discusses ramifications for cloud computing paradigms, and Chapter 6 examines deceptive moving-target defense.

1.4 DECEPTION AND OBSCURITY

One reason some researchers have neglected or eschewed deception as a basis for computer security is its similarity to obscurity. "Security through obscurity"[1] has become a byword for security practices that rely upon an adversary's ignorance of the system design rather than any fundamental principle of security. History has demonstrated that such practices offer very weak security at best, and are dangerously misleading at worst, potentially offering an illusion of security that may encourage poor decision-making.[2]

[1] *cf., Merkow and Breithaupt (2014)*

[2] *Anderson (2001)*

Security defenses based on deception potentially run the risk of falling into the "security through obscurity" trap. If the defense's deceptiveness hinges on attacker ignorance of the system design—details that defenders should conservatively assume will eventually become known by any suitably persistent threat actor—then any security offered by the defense might be illusory and therefore untrustworthy. It is therefore important to carefully examine the underlying basis upon which deceptions can be viewed as security-enhancing technologies.

Using honey-patching as an example, the effectiveness of a patching-based deception relies upon withholding certain secrets from adversaries (e.g., exactly which software vulnerabilities have been honey-patched). But secret-keeping does not in itself disqualify honey-patching as obscurity-reliant. For example, modern cryptography is frequently championed as a hallmark of anti-obscurity defense despite its foundational assumption that adversaries lack knowledge of private keys, because disclosing the complete implementation details of crypto algorithms does not aid attackers in breaking cyphertexts derived from undisclosed keys.

Juels[1] defines *indistinguishability* and *secrecy* as two properties required for successful deployment of honey systems. These properties are formalized as follows:

[1] *Juels (2014)*

> Consider a simple system in which $S = \{s_1, \ldots, s_n\}$ denotes a set of n objects of which one, $s^* = s_j$, for $j \in \{1, \ldots, n\}$ is the true object, while the other $n - 1$ are honey objects. The two properties then are:
>
> *Indistinguishability*: To deceive an attacker, honey objects must be hard to distinguish from real objects. They should, in other words, be drawn from a probability distribution over possible objects similar to that of real objects.
>
> *Secrecy*: In a system with honey objects, j is a secret. Honey objects can, of course, only deceive an attacker that doesn't know j, so j cannot reside alongside S. Kerckhoffs' principle therefore comes into play: the security of the system must reside in the secret, i.e., the distinction between honey objects and real ones, not in the mere fact of using honey objects.

Honey-patching as a paradigm satisfies both these properties by design:

Indistinguishability derives from the inability of an attacker to determine whether an apparently successful attack is the result of exploiting an unpatched vulnerability or a honey-patch masquerading as an unpatched vulnerability. While absolute, universal indistinguishability is probably impossible to achieve, many forms of distinguishability can nevertheless be made arbitrarily difficult to discern. For example, honey-servers can exhibit response delay distributions that mimic those of unpatched servers to arbitrary degrees of precision (e.g., by artificially delaying legitimate, non-forking requests to match the distribution of malicious, forking requests, as described in Section 3.4).

Secrecy implies that the set of honey-patched vulnerabilities should be secret. However, full attacker knowledge of the design and implementation details of honey-patching does not disclose *which* vulnerabilities a defender has identified and honey-patched. Adapting Kerckhoffs' principle[1] for deception, a honey-patch is not detectable even if everything about the system, except the honey-patch, is public knowledge.

[1] *Kerckhoffs (1883)*

This argues that honey-patching as a paradigm (and language-based software cyber deception in general) does not derive its security value from obscurity. Rather, its deceptions are based on well-defined secrets—specifically, the set of honey-patched vulnerabilities in target applications. Maintaining this confidentiality distinction between the publicness of honey-patch design and implementation details, versus the secrecy of exactly which vulnerabilities instantiate those details, is important for crafting robust, effective deceptions.

Motivationally, the difference between obscurity-based defenses and cyber deceptions as advanced in this dissertation can be seen as a difference in defender objective: The objective of an obscurity-based defense is to create uncertainty in the mind of an adversary; whereas the objective of a deception-based defense is to inspire *false certainty*. Each of the deceptive technologies introduced herein therefore cultivate digital environments conducive to incorrect conclusions on the part of attackers, rather than merely to cultivate environments that are bewildering or foggy.

1.5 DISSERTATION OVERVIEW

The rest of this dissertation is laid out as follows. Chapter 2 details honey-patching and its underlying foundations. Chapter 3 defines the associated problem of sanitizing secrets from process images, and outlines a solution based on compiler-instrumented, dynamic taint analysis. Chapters 4–6 present a suite of security defenses leveraging this framework: (1) DeepDig leverages attack traces collected via honey-patches to generate models of attacker behavior and enhance precision of anomaly detection (Chapter 4). (2) Deception-as-a-Service is proposed as a new paradigm to transparently enable honey-patching in service-oriented architectures (Chapter 5). (3) QUICKSAND implements a moving target architecture to render honey-patched applications more robust against deception fingerprinting techniques (Chapter 6). Chapter 7 documents experiences using honey-patching in the classroom. Finally, relevant related work is presented in Chapter 8 and conclusions are presented in Chapter 9.

2 HONEY-PATCHING

When a software security vulnerability is discovered, the conventional defender reaction is to quickly *patch* the software to fix the problem. This standard reaction can backfire, however, if the patch has the side-effect of disclosing and highlighting other exploitable weaknesses in the defender's network. Unfortunately, such backfires are common; patches often behave in such a way that adversaries can reliably infer which systems have been patched, and therefore which are unpatched and vulnerable. The existence of at least some unpatched systems is almost inevitable, since patch adoption is rarely immediate—for example, testing is often required to ensure patch compatibility. Thus, most software security patches fix newly discovered vulnerabilities at the price of advertising to attackers which systems remain vulnerable. This has led to an adversarial culture for which *vulnerability probing* is a staple of the cyber killchain.

This chapter contains material previously published as: F. Araujo, K.W. Hamlen, S. Biedermann, and S. Katzenbeisser. "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation." In Proc. of the 21st ACM SIGSAC Conf. on Computer and Communications Security, pp. 942–953, Nov. 2014. Lead author Araujo conducted the majority of the research, including the design, most of the implementation, and the evaluation of the prototype.

Cyber criminals easily probe today's Internet for vulnerable software, allowing them to focus their attacks on susceptible targets, in the following way. First, the attacker submits a malicious input (a *probe*) crafted to trigger a particular, known software bug in bulk to many servers across the network. Patched servers respond to the probe with a well-formed output, such as an error message; but unpatched servers behave erratically, such as by responding with a garbage string or crashing and restarting. Upon observing the latter response, the attacker next submits a more constructive malicious input to the unpatched servers, such as one that exploits the bug to hijack the victim software's control-flow, causing it to perform malicious actions on behalf of the attacker rather than merely crashing.

[1] Araujo et al. (2014)

Honey-patching[1] is a game-changing alternative approach for anticipating and foiling these directed cyber attacks. The goal is to patch newly discovered software security vulnerabilities in such a way that future attempted exploits of the patched vulnerabilities appear successful to attackers even when they are not. This masks patching lapses, impeding attackers from easily discerning which systems are genuinely vulnerable and which are actually patched systems masquerading as unpatched systems. Detected attacks are transparently redirected to isolated, unpatched decoy environments that possess the full interactive power of the targeted victim server, but that disinform adversaries with honey-data and aggressively monitor adversarial behavior.

Deceptive honey-patching capabilities thereby constitute an advanced, language-based, active defense technique that can impede, confound, and misdirect attacks, and significantly raise attacker risk and uncertainty. In addition to helping protect networks where honey-patches are deployed, the practice also contributes to the public cyber welfare: Once a honey-patch for a particular software vulnerability has been adopted by some, attacks against all networks become riskier for attackers. This is because attackers can no longer reliably identify all the vulnerable systems and determine where to focus their attacks, or even assert whether they are gathering genuine data. Any ostensibly vulnerable network could be a honey-patch in disguise, and any exfiltrated secret could potentially be disinformation.

2.1 OVERVIEW

[2] Codenomicon (2014)

Listing 2.1 shows an abbreviated patch in diff style for the Heartbleed OpenSSL buffer over-read vulnerability (CVE-2014-0160)[2]—one of the most significant vulnerability disclosures in recent history, affecting a majority of then-deployed

Listing 2.1: Abbreviated patch for Heartbleed

```
+ if (1 + 2 + payload + 16 > s→s3→rrec.length)
+ {
+       return 0;  // silently discard
+ }
```

Listing 2.2: Honey-patch for Heartbleed

```
  if (1 + 2 + payload + 16 > s→s3→rrec.length)
+ {
+       hp_fork();
-       return 0;  // silently discard
+       hp_skip(return 0);  // silently discard
+ }
```

web servers, including Apache. The patch introduces a conditional that validates SSL/TLS heartbeat packets, declining malformed requests. Prior to being patched, attackers could exploit this bug to acquire sensitive information from many web servers.

This patch exemplifies a common vulnerability mitigation: dangerous inputs or program states are detected via a boolean test, with positive detection eliciting a corrective action. The corrective action is typically readily distinguishable by attackers—in this case, the attacker request is silently declined. As a result, the patched and unpatched programs differ only on attack inputs, making the patched system susceptible to probing. Our goal in this work is to introduce a strategy whereby administrators of products such as Apache can easily transform such patches into honey-patches, whose corrective actions impede attackers and offer strategic benefits to defenders.

Toward this end, Listing 2.2 presents an alternative, honey-patched implementation of the same patch. In response to a malformed input, the honey-patched application forks itself onto a confined, ephemeral, decoy environment, and behaves henceforth as an unpatched, vulnerable version of the software. Specifically, line 3 forks the user session to a decoy container, and macro `hp_skip` in line 5 elides the rejection in the decoy container so that the attack appears to have succeeded. Meanwhile, the attacker session in the original container is safely terminated (having been forked to the decoy), and legitimate, concurrent connections continue unaffected.

Observe that the differences between the patch and the honey-patch are quite minor, except for the fixed cloning infrastructure that the honey-patch code references, and that can be maintained separately from the server code. This allowed us to formulate a Heartbleed honey-patch within hours of receiving the vulnerability disclosure on April 7, 2014, facilitating a quick, aggressive response to the threat.[1] In general, only a superficial understanding of many patches is required to convert them to honey-patches of this form. (A more systematic study of honey-patchable patches is presented in Section 2.5.) However, the cloning infrastructure required to facilitate efficient, transparent, and safe redirection to decoys demands a careful design.

[1] *The Economic Times (2014)*

2.1.1 *Design Principles*

Although the honey-patching concept is fairly straightforward, many significant security and performance challenges must be surmounted to realize it in practice. For example, a honey-patch that naïvely forks the entire server process to create a decoy clone process in response to attempted intrusions, inadvertently copies any secrets in the victim process's address space, such as encryption keys of concurrent sessions, over to the child decoy. Such an approach would be disastrous in practice, since the attack is allowed to succeed in the decoy, thereby giving the attacker potential access to secrets it may contain.

Moreover, practical adoption requires that honey-patches (1) introduce almost no overhead for legitimate users, (2) perform well enough for attackers that attack failures are not placarded, and (3) offer high compatibility with software that boasts aggressive multi-processing, multi-threading, and active connection migration across IPs. Solutions must therefore be sufficiently modular and generic that administrators require only a superficial, high-level understanding of each patch's structure and semantics to reformulate it as an effective honey-patch.

Specifically, effective honey-patching requires that remote forking of attacker sessions to decoys must happen live, with no perceptible disruption in the target application. This means that established connections—in particular, the attacker's connection—must not be broken. In addition, decoy deployment must be fast, to avoid offering overt, reliable timing channels that advertise the honey-patch. Finally, all sensitive data must be redacted before the decoy resumes execution to avoid giving the attacker potential access to user secrets.

APPROACH Together, these requirements motivate three main design decisions. First, the required time performance precludes system-level cloning (e.g., VM cloning[1]) for session forking; instead, we employ a lighter-weight, finer-grained alternative based on process migration through *checkpoint-restart*.[2] To scale to many concurrent attacks, we use an *OS-level virtualization* technique to deploy forked processes to decoy containers, which can be created, deployed, and destroyed orders of magnitude faster than other virtualization techniques, such as full virtualization or para-virtualization.[3]

[1] *Clark et al. (2005)*

[2] *Milojičić et al. (2000)*

[3] *Whitaker et al. (2004)*

Second, our approach to remote session forking benefits from the synergy between mainstream Linux kernel APIs and user-space tools, allowing for a small freezing time of the target application. To maintain established connections when forking, we have conceived and implemented a *connection relocation* procedure that allows for transparent session migration.

Third, to guarantee that successful exploits do not afford attackers access to sensitive data stored in application memory, we have implemented a *memory redaction* and light-weight synchronization mechanism during forking. This censors sensitive data from process memory before the forked (unpatched) session resumes. Forked decoys host a deceptive file system that omits all secrets, and that can be laced with disinformation to further deceive, delay, and misdirect attackers.

2.1.2 *Threat Model*

Honey-patches add a layer of deception to confound exploits of known (patchable) vulnerabilities, which constitute the majority of exploited vulnerabilities in the wild. Previously unknown (i.e., zero-day) exploits remain potential threats, since such vulnerabilities are typically neither patched nor honey-patched. However, even zero-days can be potentially mitigated through cooperation of honey-patches with other layers of the deception stack. For example, a honey-patch that collects identifying information about a particular adversary seeking to exploit a known vulnerability can convey that collected information to a network-level intrusion detection system, which can then potentially identify the same adversary seeking to exploit a previously unknown vulnerability. This strategy is explored in Chapter 4.

Although honey-patches primarily mitigate exploits of known vulnerabilities, they can effectively mitigate exploits whose attack payloads might be completely unique and therefore unknown to defenders. Such payloads might elude network-level monitors, and are therefore best detected at the software level at the point of exploit. Attackers might also use one payload for reconnaissance but reserve another for the final attack. Misleading the attacker into launching the final attack is therefore useful for discovering the final attack payload, which can divulge attacker strategies and goals not discernible from the reconnaissance payload alone.

Honey-patching is typically used in conjunction with standard access control protections, such as process isolation and least privilege. Attacker requests are therefore typically processed by a server possessing strictly user-level privileges, and must therefore leverage web server bugs and kernel-supplied services to perform malicious actions, such as corrupting the file system or accessing other users' memory to access confidential data. The defender's ability to thwart these and future attacks stems from his ability to deflect attackers to fully isolated decoys and perform counterreconnaissance (e.g., attack attribution and information gathering).

2.1.3 *Background*

Apache HTTP has been the most popular web server since April 1996.[1] Its market share includes 54.5% of all active websites (the second, Nginx, has 11.97%) and 55.45% of the top-million websites (against Nginx with 15.91%).[2] It is a robust, commercial-grade, feature-rich open-source software product comprised of 2.27M SLOC mostly in C,[3] and has been tested on millions of web servers around the world. These characteristics make it a highly challenging, interesting, and practical flagship case study to test our approach.

[1] *Apache (2014)*

[2] *Netcraft (2014)*

[3] *Ohloh (2014)*

Process migration through checkpoint-restart is the act of transferring a running process between two nodes by dumping its state on the source and resuming its execution on the destination. This problem is especially relevant for high-performance computing.[4] As a result, several tools have been developed to support performance-critical process checkpoint-restart (e.g., BLCR,[5] DMTCP,[6] and CRIU[7]). Process checkpoint-restart plays a pivotal role in making the honey-patch concept viable. It provides a fast and seamless mechanism to enable transparent forking of attacker sessions, and scales well even in small environments due to its process-level granularity, which reduces the overall resources required to migrate the attacker process.

[4] *Gerofi et al. (2010); Wang et al. (2008)*

[5] *Duell (2002)*

[6] *Ansel et al. (2009)*

[7] *CRIU (2014)*

OS-level virtualization allows multiple guest nodes (*containers*) to share the kernel of their controlling host. Linux

[1]*LXC (2014)*

containers (LXC)[1] implement OS-level virtualization, with resource management via process control groups and full resource isolation via Linux namespaces. This ensures that each container's processes, file system, network, and users remain mutually isolated. Fine-grained control of resource utilization prevents any container from starving its host. Furthermore, LXC supports containers backed by *overlayfs* snapshots, which is key for efficient container management and fast decoy deployment.

2.2 ARCHITECTURE

[2]*Araujo et al. (2014)*

The RedHerring architecture,[2] depicted in Figure 2.1, embodies these design decisions by using process-level cloning and OS-level virtualization to achieve lightweight, resource-efficient, and fine-grained redirection of attacker sessions to sandboxed decoy environments in which secrets have been redacted with honey-data. Within this framework, developers use honey-patches to provide the same level of security as conventional patches, yet have the additional ability to deceive attackers.

Central to the system is a *reverse proxy* that acts as a transparent proxy between users and internal servers deployed as LXC containers. The *target* container hosts the honey-patched web server instance, and the n decoys form the pool of ephemeral containers managed by the *LXC Controller*. The decoys serve as temporary environments for attacker sessions. Each container runs a *CR-Service* (Checkpoint/Restore) daemon, which exposes an interface controlled by the *CR-Controller* for remote checkpoint and restore.

HONEY-PATCHING API To achieve low-coupling between target application and honey-patching logic, the honey-patch mechanism can be realized as a tiny library (e.g., implemented as a dynamically loadable C library). The library exposes three core API functions:

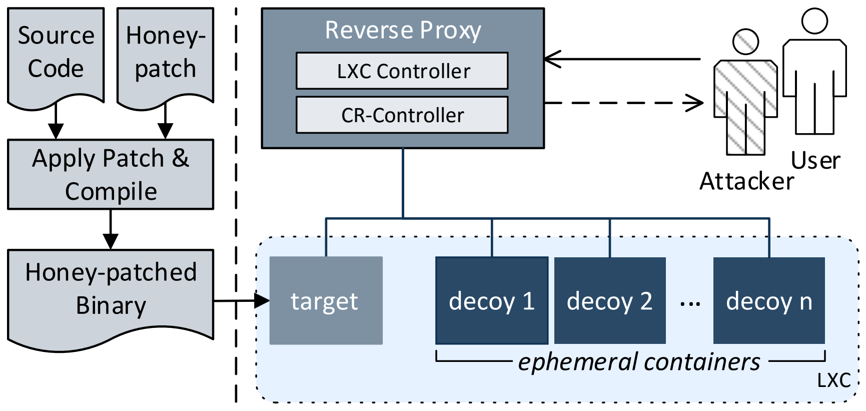


Figure 2.1: RedHerring system architecture overview

- `hp_init(`*pgid*, *pid*, *tid*, *sk*`)`: initialize honey-patch with the process group *pgid*, process *pid*, thread *tid*, and socket descriptor *sk* of the session.

- `hp_fork()`: initiate the attacker session remote forking process, implementing the honey-patching core logic.

- `hp_skip(c)`: skip over block `c` if in a decoy.

Function `hp_init` initializes the honey-patch with the necessary information to handle subsequent session termination and resurrection. It is invoked once per connection, at the start of the session life cycle. For example, in the Apache web server, this immediately follows acceptance of an HTTP request and handing the newly created session off to a child process or worker thread; in Lighttpd and Nginx web servers, it follows the accept event for new connections.

Listing 2.3 details the basic steps of `hp_fork`. Line 3 determines the application context, which can be either `target` (denoting the target container) or `decoy`. In a decoy, the function does nothing, allowing multiple exploits within a single attacker session to continue within the same decoy. In the target, a fork is initiated, consisting of four steps: (1) Line 5 registers the signal handler for session termination and resurrection. (2) Line 6 sends a *fork request* containing the attacker session's *pgid*, *pid*, and *tid* to the proxy's CR-Controller. (3) Line 7 synchronizes checkpoint and restore of the attacker session in the target and decoy, respectively, and guarantees that sensitive data is redacted from memory before the clone is allowed to resume. (4) Once forking is complete and the attacker session has been resurrected, the honey-patch context is saved and the attacker session resumes in the decoy.

The fork request (step 2) achieves high efficiency by first issuing a system `fork` to create a shallow, local clone of the web server process. This allows event-driven web servers to continue while attacker sessions are forked onto decoys, without interrupting the main event-loop. It also lifts the burden of synchronizing concurrent checkpoint operations, since CRIU injects a Binary, Large OBject (BLOB) into the target process memory space to extract state data during checkpoint.

The context-sensitivity of this framework allows the honey-patch code to exhibit context-specific behavior: In decoy contexts, `hp_skip` elides the execution of the code block passed as an argument to the macro, elegantly simulating the unpatched application code. In a target context, it is usually never reached due to the fork. However, if forking silently fails (e.g., due to resource exhaustion), it abandons the deception and conservatively executes the original patch's corrective action for safety.

LXC POOL The decoys into which attacker sessions are forked are managed as a pool of Linux containers controlled by the LXC Controller. The controller exposes two operations to the proxy: *acquire* (to acquire a container from the pool), and *release* (to release back a container to the pool). Each container follows the life cycle depicted in Figure 2.2. Upon receiving a fork request, the proxy acquires the first available container from the pool. The acquired container holds an attacker session until (1) the session is deliberately closed by the attacker, (2) the connection's *keep-alive* timeout expires, (3) the ephemeral container crashes, or (4) a session timeout

Listing 2.3: `hp_fork function`

```
void hp_fork()
{
      read_context();            // read context (target/decoy)
      if (decoy) return;         // if in decoy, do nothing
      register_handler();        // register signal handler
      request_fork();            // fork session to decoy
      wait();                    // wait until fork process has finished
      save_context();            // save context and resume
}
```

is reached. The last two conditions are common outcomes of successful exploits. In any of these cases, the container is released back to the pool and undergoes a recycling process before becoming available again.

Recycling a container encompasses three sequential operations: *destroy*, *clone* (which creates a new container from a template in which legitimate files are replaced with honeyfiles), and *start*. These steps happen swiftly for two main reasons. First, the lightweight virtualization implemented by LXC allows containers to be destroyed and started similarly to how OS processes are terminated and created. Second, ephemeral containers are deployed as overlayfs-based clones, making the cloning step almost instantaneous. The overlay file system is backed by a regular directory (the *template*) to clone new *overlayfs* containers (decoys), mounting the template's root file system as a read-only lower mount and a new private delta directory as a read-write upper mount. The template used to clone decoys is a copy of the target container in which all sensitive files are replaced with honey-files.

CR-SERVICE The Reverse Proxy uses the CR-Controller module to communicate with CR-Service daemons running in the background of each container. The CR-Service uses an extended version of CRIU (Checkpoint/Restore In Userspace)[1] to checkpoint attacker sessions on the target and restore them

[1] *CRIU (2014)*

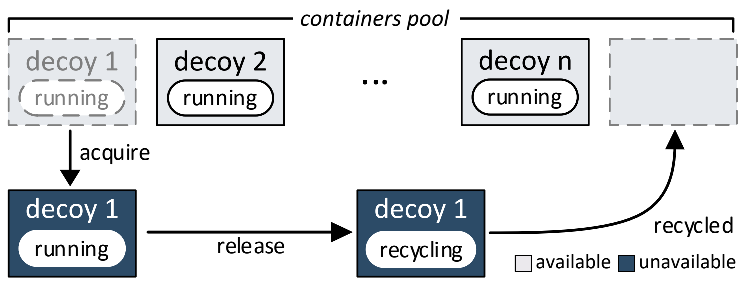


Figure 2.2: Linux containers pool and decoys life cycle

on decoys. Each CR-Service implements a *façade* that exposes CR operations to the proxy's CR-Controller through a simple RPC protocol based on Protocol Buffers.[1] To enable fast, OS-local RPC communication between proxy and containers, IPC sockets (a.k.a., Unix domain sockets) are used.

[1]*Google (2008)*

However, because IPC sockets rely on the file system as an address namespace and the proxy runs on the host, establishing cross-container connections becomes difficult: host and container file-systems are opaque to each other (due to namespace isolation). To overcome this issue, a directory located in the host is configured as bind mount to all containers to be used as bridge for IPC sockets, thus enabling the establishment of IPC connections between the CR-Controller running in the host and the CR-Service instances running inside containers.

REVERSE PROXY The proxy plays a dual role in the honey-patching system, acting as (1) a *transport layer transparent proxy*, and (2) an *orchestrator* for attacker session forking.

As a transparent proxy, its main purpose is to hide the backend web servers and route client requests. To serve each client's request, the proxy server accepts a downstream socket connection from the client and binds an upstream socket connection to the backend server, allowing application-layer sessions to be processed transparently between the client and the backend server. To keep its size small, the proxy neither manipulates message payloads, nor implements any

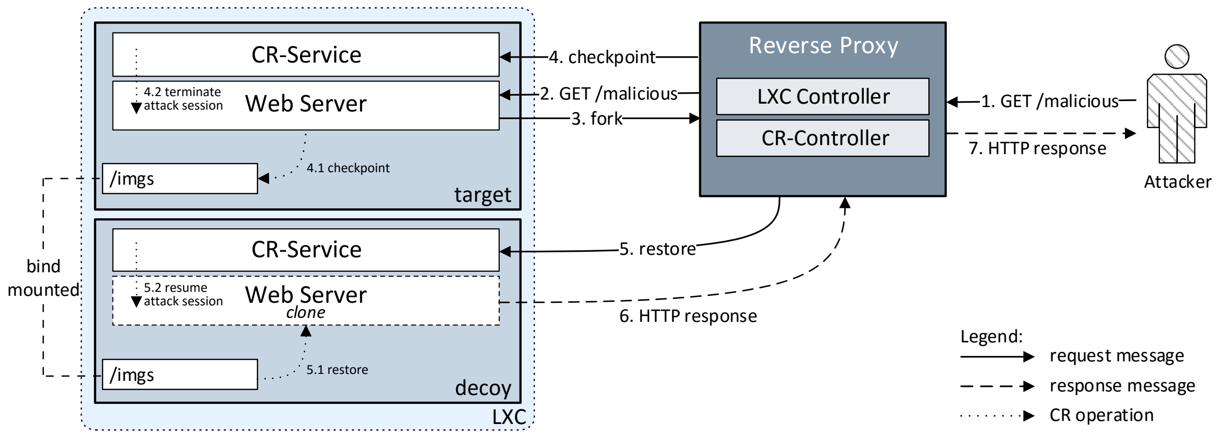


Figure 2.3: Attacker session forking. Numbers indicate the sequential steps taken to fork an attacker session.

rules for detecting attacks. There is also no session caching. This makes it extremely innocuous and lightweight. The proxy is implemented as a transport-layer reverse proxy to reduce routing overhead and support the variety of protocols operating above TCP, including SSL/TLS.

As an orchestrator, the proxy listens for fork requests and coordinates the attacker session forking as shown in Figure 2.3. Under legitimate load, the proxy simply routes user requests to the target and routes server responses to users. However, attack inputs elicit the following alternate workflow:

STEP 1: The attacker probes the server with a crafted request (denoted by malicious request `GET /malicious` in Figure 2.3).

STEP 2: The reverse proxy transparently routes the request to the backend target web server.

STEP 3: The request triggers the honey-patch (i.e., when the honey-patch detects an attempted exploit of the patched vulnerability) and issues a fork request to the reverse proxy.

STEP 4: The proxy's CR-Controller processes the request, acquires a decoy from the LXC Pool, and issues a checkpoint RPC request to the target's CR-Service. The CR-Service

4.1: checkpoints the running web server instance to the /imgs directory; and

4.2: signals the attacker session with a termination code, gracefully terminating it.

STEP 5: Upon checkpoint completion, the CR-Controller commands the decoy's CR-Service to restore the dumped web server images on the decoy. The CR-Service then

5.1: restores a clone of the web server from the dump images located in the /imgs directory; and

5.2: signals the attacker session with a resume code, and cleans the dump data from /imgs.

STEP 6: The attacker session resumes on the decoy, and a response is sent back to the reverse proxy.

STEP 7: The reverse proxy routes the response to the attacker.

Throughout this workflow, the attacker's session forking is completely transparent to the attacker. To avoid any substantial overhead for transferring files between target and decoys, each decoy's /imgs folder is bind-mounted to the target's /imgs directory. After the session has been forked to the decoy, it behaves like an unpatched server, making it appear that no redirection has taken place and the original probed server is vulnerable.

2.3 SESSION REMOTE FORKING

At the core of our architecture is the capability of remote forking an attacker session to a decoy through checkpoint and restore of the target server. To this end, we have extended CRIU[1] with a memory redaction procedure performed during checkpoint to protect sensitive data of legitimate users, and a transparent connection relocation mechanism to restore TCP connections in the destination decoy without stopping the target server. We name this extended version CRIU_m.

[1] *CRIU (2014)*

2.3.1 *Checkpoint*

The checkpoint procedure takes place in the target container and is initiated when the CR-Service receives a checkpoint request. The request includes the process group leader `$pgid`, attacker process `$pid`, and attacker thread `$tid`.

The CR-Service passes this information to our CRIU$_\mathrm{m}$ checkpoint interface, which in turn: (1) uses the /proc file system to collect file descriptors (/proc/$pgid/fd and /proc/$pgid/fdinfo), pipe parameters, and memory maps (/proc/$pgid/maps) for the process group; (2) walks through /proc/$pgid/task/ and gathers child processes recursively to build the process tree; (3) locks the network by adding netfilter rules and collecting socket information; (4) uses `ptrace` (with `PTRACE_SEIZE`) to attach to each child (without stopping it) and collect VMA areas, the task's file descriptor numbers, and core parameters such as registers; (5) injects a BLOB code into the child address space to collect state information such as memory pages; (6) performs memory redaction using `$pid` and `$tid`; (7) uses `ptrace` to remove the injected code from the child process and continues until all children have been traced; (8) unlocks network using netfilter, and finishes the procedure by writing the process tree image files to /imgs/$tid/.

At this point, CRIU$_\mathrm{m}$ returns to the caller, the web server is running, and the attacker thread waits to be signaled. The CR-Service then sends a termination signal to the attacker thread, which terminates itself gracefully in the target web server. This successfully completes the checkpoint request, and the CR-Service sends a `success` status response to the CR-Controller.

We next examine the memory redaction step in greater detail, to explain how sensitive, in-memory data is safely replaced with decoy data during the fork.

MEMORY REDACTION Were session cloning performed in the typical, rote fashion of copying all bytes, attackers who successfully hijack decoys could potentially view any confidential data copied from the memory space of the original process (e.g., in a multi-threaded setting). Sophisticated attacks could thus glean sensitive information about other users previously or concurrently connected to the original server process, if such information is cloned with the process. In web servers, such sensitive information includes IP addresses of other users, request histories, and information about encrypted connections. It is therefore important to redact these secrets during cloning.

We therefore introduce a memory redaction procedure that replaces sensitive data with specially forged, anonymous data during cloning. Since every server application has different forms of sensitive data stored in slightly different ways, our solution is a general-purpose tool that must be specialized to each server product by an administrator prior to deployment. In the case of Apache, we focus on redaction of user request data, session data, and SSL context data, which Apache records in a few data structures stored in memory for each user session. For instance, Apache's `request_rec` struct stores request histories. Other servers store such data in similar ways: Nginx stores request records into struct `ngx_http_request_s`, which references structs `ngx_http_headers_in_t` and `ngx_http_request_body_t` for request headers and body, respectively, and Lighttd stores request histories into structs `request`, `request_uri`, and `response`.

Without special compiler-side support for the redaction process, redaction can be implemented as a memory sweep that heuristically identifies secrets and replaces them with honey-data at session forking time. This chapter reports the implementation and evaluation of a sweep-style redactor of this sort. Chapter 3 subsequently introduces a compiler-side technology based on dynamic taint tracking that is more precise and more efficient, but that requires compiler support.

A brute force strategy for sweep-style memory redaction is to search the entire process memory space to match and replace sensitive data. Such a strategy does not perform well.

Instead, we leverage the fact that most security-relevant data are stored in struct variables in heap or stack memory, allowing us to narrow the search space significantly. Freed memory is included in the search. For efficiency, our redactor replaces these structures with anonymous data having exactly the same length and characteristics. For example, IP addresses in `request_rec` are replaced with strings having the same length that are also valid IP addresses, but randomly generated. This yields a realistic, consistent process image that can continue running without errors (save possibly for effects of the attack).

The redaction is implemented as a step of the checkpoint procedure, so that the image files temporarily created during process checkpoint and shared with decoys do not contain any sensitive information that could be potentially abused by attackers. Secrets are redacted from all session-specific structures except the attacker's, allowing the attacker's session to continue uninterrupted.

We initially implemented memory redaction as a separate operation applied to the image files generated by CRIU. While this seemed attractive for avoiding modification of CRIU, it exhibited poor performance due to reading and writing the image files multiple times. Our revised implementation therefore realizes redaction as a streaming operation within CRIU's checkpointing algorithm. In-lining it within checkpointing avoids reloading the process tree images into memory for redaction. In addition, redacting secrets before dumping the process images avoids ever placing secrets on disk.

2.3.2 *Restore*

Upon successful completion of a checkpoint operation, the CR-Controller sends a request to the decoy's CR-Service into which the attacker session is to be forked. In addition to `$pgid`, `$pid`, and `$tid`, the body of the restore request contains a callback `port` that has been dynamically assigned by the reverse proxy to hold the new back-end connection associated with the attacker session. Once the request is parsed, the CR-Service passes this information as parameters to the CRIU$_m$

restore interface, which (1) reads the corresponding process tree from `/imgs/$tid/`; (2) uses the `clone` system call to start each dumped process found in the process tree with its original process ID; (3) restores file descriptors and pipes to their original states, and executes relocation of `ESTABLISHED` socket connections; (4) injects a BLOB code into the process address space to recreate the memory map from the dumped data; (5) removes the injected binary, and resumes the execution of the application via the `rt_sigreturn` system call.

At this point, CRIU_m returns to the caller, the forked instance is running on the decoy, and the attacker thread waits to be signaled. The CR-Service sends a resume signal to the attacker thread, which allows it to resume request processing. This completes the restore request, and the CR-Service sends a `success` response to the CR-Controller. Subsequent attacker requests are relayed to the decoy instead of the target, as discussed in Section 2.2. Next, we discuss details of the TCP connection relocation procedure.

ESTABLISHED TCP CONNECTION RELOCATION Target and decoys are fully isolated containers running on separate namespaces. As a result, each container is assigned a unique IP in the internal network, which affects how active connections are moved from the target to a decoy. To realize this use case, an extension to CRIU is implemented as part of the honey-patching framework to support relocation of TCP connections during process restoration. In what follows, we will discuss important details of the implementation.

The reverse proxy always routes legitimate user connections to the target; hence, there is no need to restore the state of connections for these users when restoring the web server on a decoy. Legitimate connections can simply be restored to *drainer sockets*, since we have no interest in maintaining legitimate user interaction with the decoys. This ensures that the associated user sessions are restored to completion without interrupting the overall application restoration.

Conversely, the attacker connection must be restored to its dumped state when switching the attacker session to a

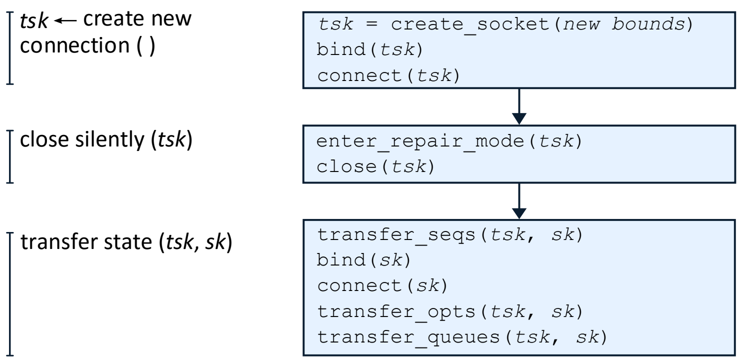


Figure 2.4: Procedure for TCP connection relocation

decoy. This is important to avoid connection disruption and to allow transparent session migration (from the perspective of the attacker). To accomplish this, the proxy dynamically establishes a new backend TCP connection between proxy and decoy containers in order to hold the attacker session communication. Moreover, a mechanism based on *TCP repair options*[1] is employed to transfer the state of the original attacker's session socket (bound to the target IP address) into the newly created socket (bound to the decoy IP address).

[1]*Corbet (2012)*

Figure 2.4 describes the connection relocation mechanism, implemented as a step of the attacker's session restore process. At process checkpoint, the state information of the original socket `sk` is dumped together with the process image (not shown in the figure). This includes connection bounds, previously negotiated socket options, sequence numbers, receiving and sending queues, and connection state. During process restore, the connection is relocated to the assigned decoy by (1) connecting a new socket `tsk` to the proxy `$port` given in the restore request, (2) setting `tsk` to *repair mode* and silently closing the socket (i.e., no `FIN` or `RST` packages are sent to the remote end), and (3) transferring the connection state from `sk` to `tsk` in repair mode. Once the new socket `tsk` is handed over to the restored attacker session, the relocation process has completed and communication resumes, often with an HTTP response being sent back to the attacker.

During connection relocation process, remote packets must not be allowed to enter the target application stack; otherwise, it would become impossible to reliably restore connections, since local and remote endpoints would reach inconsistent states (e.g., due to sequence number mismatch). Established connections are therefore *locked* during relocation, leveraging a simple netfilter rule that is configured in the target to drop all packets from remote endpoints, while retransmission of dropped packets is delegated to conventional TCP handling. This guarantees that the state of remote sockets remain consistent with their local counterparts while the attacker session is being forked over a decoy.

2.4 IMPLEMENTATION

We have developed an implementation of RedHerring for the 64-bit version of Linux (kernel 3.11 or above). The implementation consists of five components: the honey-patch library, the LXC-Controller, the CR-Controller, the CR-Service, and the reverse proxy. The honey-patch library provides the tiny API required for triggering the honey-patching mechanism. Its implementation consists of about 270 lines of C code that uses no external libraries or utilities. The reverse proxy routes HTTP/S requests in accordance with the behavior described in Section 2.2. Its implementation is fully asynchronous and consists of about 325 lines of node.js JavaScript code. The CR-Controller is implemented as an external C++ module to the proxy, and consists of about 450 lines of code that uses Protocol Buffers to communicate with the CR-Service. Similarly, the LXC-Controller is implemented as an external node.js library consisting of about 190 lines of code. The CR-Service receives CR requests from the CR-Controller and uses CRIU_m to coordinate process checkpoint and restore. Its implementation comprises about 525 lines of C code. Our extensions to CRIU_m add about 710 lines of C code to the original CRIU tool.

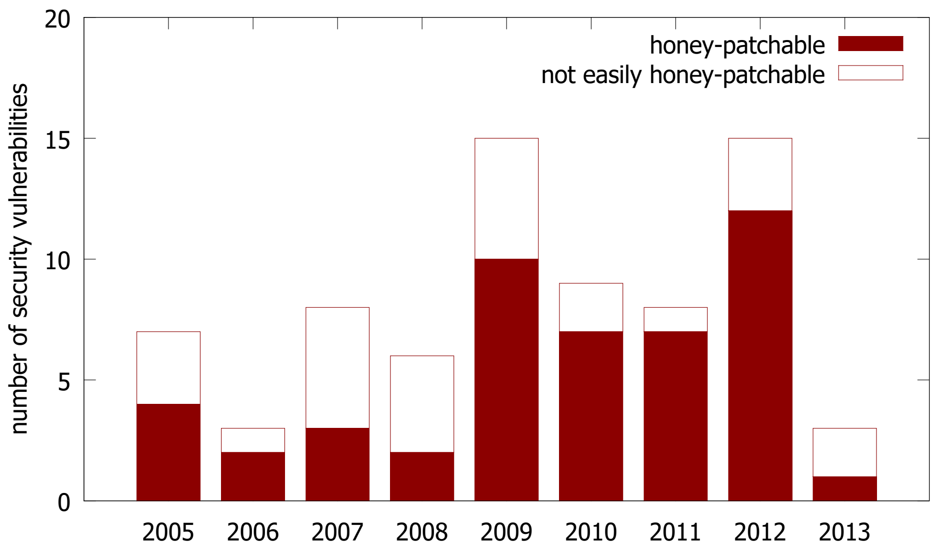


Figure 2.5: Apache honey-patchable vulnerabilities

2.5 EVALUATION

This section discusses the applicability of honey-patching and investigates performance characteristics of the session live migration scheme implemented by RedHerring. First, we survey the past nine years of Apache's security reports to assess the proportion of security patches that are amenable to our honey-patching technique. Then, we investigate the effect of session migration on malicious attack HTTP response times and report measurements of the impact of concurrent attacks on legitimate HTTP request round-trip times. Finally, we compare the performances of the honey-patched versions of Apache, Lighttpd, and Nginx.

All experiments were performed on a quad-core virtual machine (VM) with 8 GB RAM running 64-bit Ubuntu 14.04 (Trusty Tahr). Each LXC container running inside the VM was created using the official LXC Ubuntu template. We limited resource utilization on decoys so that a successful attack does not starve the host VM. The host machine is an Intel Xeon E5645 desktop running 64-bit Windows 7.

2.5.1 *Honey-patchable Patches*

Our strategy sketched in Section 2.1 for transforming patches into honey-patches is more easily applied to some patches than others. In general, patches that have a clear, boolean decision point where patched and unpatched application behavior diverge are best suited to our approach, whereas patches that introduce deeper changes to the application's control-flow structure or data structures may require correspondingly deeper knowledge of the patch's semantics to reformulate as a honey-patch.

To assess the practicality of honey-patching, we surveyed all vulnerabilities officially reported by the Apache HTTP web server project between 2005 and 2013. We systematically examined each security patch file and corresponding source code to determine its amenability to honey-patching. Figure 2.5 reports the results. Overall, we found that 49 out of 75 patches analyzed (roughly 65%) are easily transformable into honey-patches. This corroborates the intuition that most security vulnerabilities are patched with some small check, usually one that performs input validation.[1]

[1] *Brumley et al. (2008)*

Listing 2.4 shows an example of a patch (simplified for brevity) for which honey-patching is not elementary. The patch replaces the insecure method `ap_get_server_name` with an alternate one (`ap_escape_logitem`) that performs input sanitization. The sanitization step lacks any boolean decision point where exploits are detected; it instead performs a string transformation that replaces dangerous inputs with non-dangerous ones. Thus, it is not obvious where to position the forking operation needed for a honey-patch.

However, even in the case of Listing 2.4, we note that honey-patching is still possible, given a sufficiently comprehensive understanding of the patch's semantics. In particular, this patch could be converted to a honey-patch by retaining both the sanitizing and non-sanitizing implementations and comparing the resulting strings. If the strings differ, the honey-patch forks the session to a decoy. Note that not every input sanitization patch can be honey-patched in this way, since

Listing 2.4: Abbreviated patch for CVE-2013-1862

```
  logline = apr_psprintf(r→pool, ...,
                ...
-               ap_get_server_name(r),
+               ap_escape_logitem(r→pool, ...(r)),
                ...
```

Table 2.1: Honey-patched security vulnerabilities for different versions of Apache

Version	CVE-ID	Description
2.2.21	CVE-2011-3368	Improper URL validation
2.2.9	CVE-2010-2791	Improper timeouts of keep-alive connections
2.2.15	CVE-2010-1452	Bad request handling
2.2.11	CVE-2009-1890	Request content length out of bounds
2.0.55	CVE-2005-3357	Bad SSL protocol check

some sanitization procedures modify even non-dangerous inputs. Thus, patches of this sort were conservatively classified as not easily honey-patchable in our study, since they require greater effort to honey-patch.

EXPERIMENTAL VALIDATION To evaluate honey-patching's effectiveness in diverting attackers to decoys, we tested RedHerring with different honey-patched Apache releases. Table 2.1 summarizes the tested versions of Apache and corresponding vulnerabilities that we successfully exploited. For each vulnerability, we tested the system on non-malicious inputs and verified that RedHerring does not fork any attacker sessions. Then we exploited honey-patched vulnerabilities, and verified that the system behaves as a vulnerable decoy server in response to the attack inputs.

Apache 2.2.21 allows the inadvertent exposure of internal resources to remote users who send specially crafted requests (CVE-2011-3368). For example, the malicious request `GET @private.com/topsecret.pdf HTTP 1.1` may result in an exposure of unpatched servers. The security patch for this vulnerability modifies `protocol.c` to send an HTTP 400 response if the request URI is not an absolute path. Our honey-patch forks to a decoy instead.

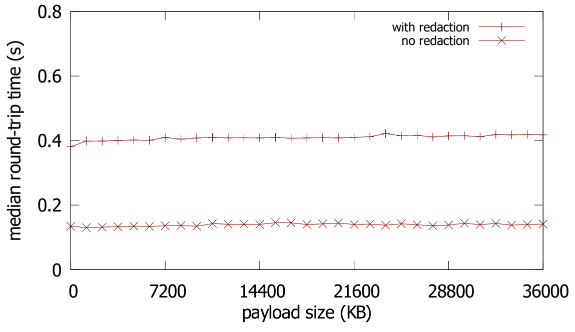
Similarly, we honey-patched and tested CVE-2010-1452 and CVE-2009-1890, which involve improper HTTP request sanitization. CVE-2010-1452 exposes a request handling problem in which requests missing the `path` field may cause the worker process to segfault, inviting potential DOS attacks. CVE-2009-1890 exposes another type of DOS vulnerability in which a sufficiently long HTTP request may lead to memory exhaustion.

CVE-2010-2791 allows us to test RedHerring against attacks exploiting vulnerabilities related to keep-alive connections. In this particular case, a bug neglects closing the back-end connection if a timeout occurs when reading a response from a persistent connection, which allows remote attackers to obtain a potentially sensitive response intended for a different client. Finally, CVE-2005-3357 exposes a bad SSL protocol check that allows an attacker to cause a DOS if a non-SSL request is directed to an SSL port.

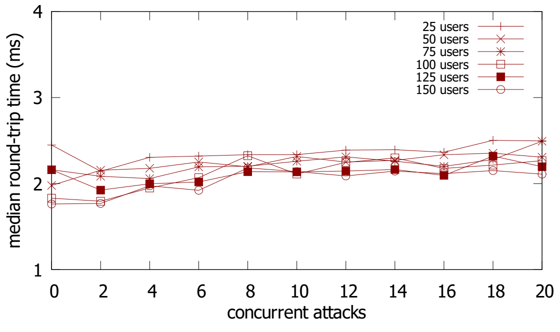
2.5.2 *Performance Benchmarks*

This section evaluates RedHerring's performance. Our objectives are two-fold: to determine the performance overhead imposed upon sessions forked to decoys (i.e., the impact on malicious users), and to estimate the impact of honey-patching on the overall system performance (i.e., its impact on legitimate users). To obtain baseline measurements that are independent of networking overhead, the experiments in this section are executed locally on a single-node virtual machine using default Apache settings. Performance is measured in terms of HTTP request round-trip time.

SESSION FORKING OVERHEAD As expected, forking attacker sessions from target to decoy containers is the main



(a) $3 \leqslant \sigma \leqslant 15$ ms, $n = 10$



(b) $0.6 \leqslant \sigma \leqslant 35$, $n = 10$

Figure 2.6: Performance benchmarks. (a) Effect of payload size on malicious HTTP request round-trip time. (b) Effect of concurrent attacks on legitimate HTTP request round-trip time on a single-node VM.

source of performance overhead in RedHerring. To estimate its impact on attacker response times, we crafted and sent malicious requests to the server in order to trigger its internal honey-patching mechanism, and measured the request round-trip time of each individual request. For accuracy, we waited for the completion of each request before sending another one.

Since Apache allocates requests (including their content) on the heap, payload size directly impacts the amount of data to be dumped in each checkpoint operation. It is therefore important to experiment using varying sizes of malicious HTTP request payload data (from 0 KB to 36 KB, in steps of 1.2 KB). Also, to estimate the response time overhead incurred from the memory redaction process, we executed our tests twice, with memory redaction enabled and disabled.

Figure 2.6a shows the encouraging results of this experiment. Malicious HTTP request round-trip times tend to remain almost constant as payload size increases. This desirable relationship can be explained by two reasons. First, CRIU's approach of copying process memory pages into dump files during checkpoints is extremely efficient. It involves a direct copy of data between file descriptors in kernel space using the `splice` system call. As a consequence, the target memory pages are never buffered into user space. Second, our approach to memory redaction leverages the fact that Apache stores session data in well-defined structs (avoiding having multiple copies in memory) to locate and redact it directly into the dump images while they are being generated by the checkpoint process. Our initial efforts to implement redaction after checkpointing exhibited far poorer performance, leading to this more efficient solution.

Overall, the round-trip times of malicious HTTP requests incur a constant overhead of approximately 0.25 seconds due to memory redaction. (When memory redaction is used, the average request takes approximately 0.40 seconds; but when disabled, it takes 0.15 seconds.) While possibly significant (depending on networking latencies), we emphasize that this constant overhead *only impacts malicious users*, as demonstrated by the next experiment.

OVERALL SYSTEM OVERHEAD To complete our evaluation, we tested RedHerring on a wide variety of workload profiles consisting of both legitimate users and attacker sessions on a single node. In this experiment, we wrote a small Python script modeling every user and attacker as a separate worker thread triggering legitimate and malicious HTTP requests, respectively. We chose the request payload size to be 2.4 KB, based on the median of KB per request measured by Google web metrics.[1] To simulate different usage profiles, we tested our system with 25–150 concurrent users, with 0–20 attackers.

[1] *Google (2014)*

Figure 2.6b plots our results. Observe that for the various profiles analyzed, the HTTP request round-trip times remain approximately constant (ranging between 1.7 and 2.5 milliseconds) when increasing the number of concurrent malicious requests. This confirms that adding honey-patching capabilities has negligible performance impact on legitimate requests and users relative to traditional patches, even during concurrent attacks. It also confirms our previous claims

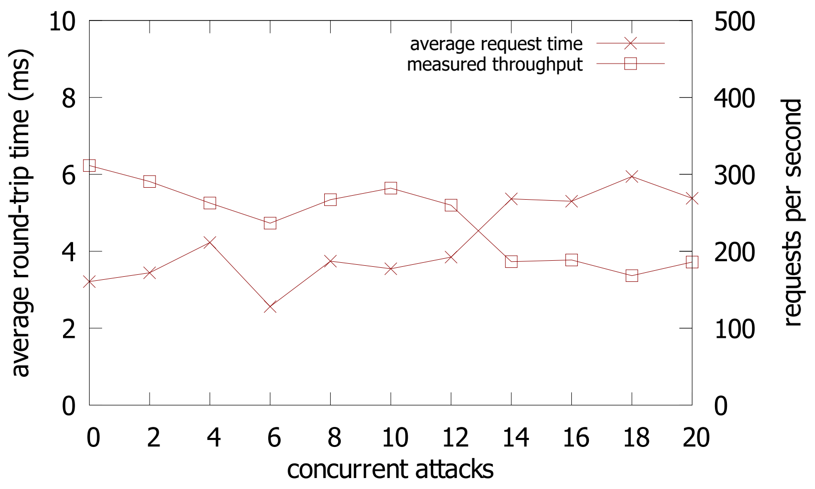


Figure 2.7: Stress test illustrating request throughput for a 3-node, load-balanced RedHerring setup (workload $\approx 5\text{K}$ requests, $0.3 \leqslant \sigma \leqslant 1.2$, $25 \leqslant \sigma \leqslant 94$)

regarding the small freezing window necessary to checkpoint the target application.

Finally, this also shows that RedHerring can cope with large workloads. In this experiment, we have assessed its baseline performance considering only one instance of the target server running on a single node virtual machine. In a real setting, we can deploy several similar instances using a web farm scheme to scale up to thousands of users, as we show next.

STRESS TESTING To estimate the throughput of our system and test its scalability properties, we developed a small HTTP load balancer in node.js to round-robin requests between three-node VMs, each hosting one instance of Apache deployed on RedHerring. In this experiment, we used *ab* (Apache HTTP server benchmarking tool) to create a massive workload of legitimate users (more than 5,000 requests in 10 threads) for different attack profiles (0 to 20 concurrent attacks). Each VM is configured with a 2 GB RAM and one quad-core processor. The load balancer and the benchmark tool run on a separate

VM on the same host machine. Apache runs with default settings (i.e., no fine tuning has been performed).

As Figure 2.7 illustrates, the system can handle the strenuous workload imposed by our test suite. The average request time for legitimate users ranged from 2.5 to 5.9 milliseconds, with measured throughput ranging from 169 to 312 requests per second. In typical production settings we would expect this delay to be amortized by the network latency (usually on the order of several tens of milliseconds). This result is important because it demonstrates that honey-patching can be realized for large-scale, performance critical software applications with minimal overheads for legitimate users.

2.5.3 *Web Servers Comparison*

We also tested RedHerring on Lighttpd[1] and Nginx,[2] web servers whose designs are significantly different from Apache. The most notable difference lies in the processing model of these servers, which employs non-blocking systems calls (e.g., select, poll, epoll) to perform asynchronous I/O operations for concurrent processing of multiple HTTP requests. In contrast, Apache dispatches each request to a child process or thread.[3] Our success with these three types of server evidences the versatility of our approach.

[1] *Lighttpd (2014)*

[2] *Nginx (2014)*

[3] *Pai et al. (1999)*

Figure 2.8 shows our results. In comparison to Apache, session forking performed considerably better on Lighttpd and Nginx (ranging between 0.092 seconds without memory redaction and 0.156 seconds with redaction). This is mainly because these servers have smaller process images, reducing the amount of state to be collected and redacted during checkpointing.

2.6 CONCLUSION

In this chapter we proposed, implemented, and evaluated honey-patching as a strategy for elegantly reformulating many

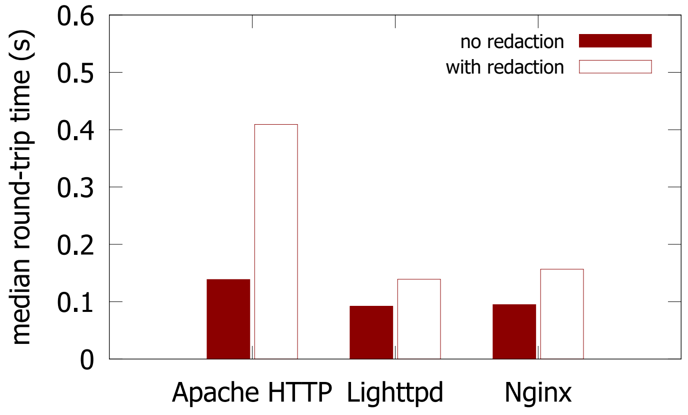


Figure 2.8: Malicious HTTP request round-trip times for different web servers ($6 \leqslant \sigma \leqslant 11$ ms, $n = 20$)

vendor-supplied, source-level patches as equally secure honey-patches that raise attacker risk and uncertainty. A light-weight, resource-efficient, and fine-grained implementation approach based on live cloning transparently forks attacker connections to sandboxed decoy environments in which in-memory and file system secrets have been redacted or replaced with honey-data. Our implementation and evaluation for the Apache HTTP web server demonstrate that honey-patching can be realized for large-scale, performance-critical software with minimal overheads for legitimate users. If adopted on a wide scale, we conjecture that honey-patching could significantly impede certain attacker activities, such as vulnerability probing, and offers defenders a new, potent tool for attacker deception.

3

PROCESS IMAGE SECRET REDACTION

This chapter contains material previously published as: F. Araujo and K.W. Hamlen. "Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception." In Proc. of the 24th USENIX Security Sym., pp. 145–159, Aug. 2015. Lead author Araujo conducted the majority of the research, including most of the design, the full implementation, and the full evaluation.

Redaction of sensitive information from documents has been used since ancient times as a means of concealing and removing secrets from texts intended for public release. As early as the 13th century B.C., Pharaoh Horemheb, in an effort to conceal the acts of his predecessors from future generations, so thoroughly located and erased their names from all monument inscriptions that their identities weren't rediscovered until the 19th century A.D.[1] In the modern era of digitally manipulated data, *dynamic taint analysis*[2] has become an important tool for automatically tracking the flow of secrets (*tainted data*) through computer programs as they execute. Taint analysis has myriad applications, including program vulnerability detection,[3] malware analysis,[4] test set generation,[5] and information leak detection.[6]

[1] *Epigraphic Survey (1994, 1998)*

[2] *cf., Schwartz et al. (2010)*

[3] *Portokalidis et al. (2006); Xu et al. (2006); Nguyen-tuong et al. (2005); Cheng et al. (2006); Chang et al. (2008); Suh et al. (2004); Newsome and Song (2005); Bosman et al. (2011); Ho et al. (2006)*

[4] *Egele et al. (2012); Yin et al. (2007); Egele et al. (2007); Papagiannis et al. (2011)*

[5] *Attariyan and Flinn (2010); Sezer et al. (2007)*

[6] *Enck et al. (2014); Zhu et al. (2011); Gibler et al. (2012); Bauer et al. (2015); Cox et al. (2014); Gu et al. (2013)*

Our research introduces and examines the associated challenge of secret redaction from program process images. Safe, efficient redaction of secrets from program address spaces has many applications, including the safe release of program memory dumps to software developers for debugging purposes, mitigation of cyber-attacks via runtime self-censoring in response to intrusions, and attacker deception through honeypotting.

Chapter 2 instantiates the latter, proposing honey-patching as a means of crafting software security patches in such a way that future attempted exploits of the patched vulnerabilities appear successful to attackers. This frustrates attacker vulnerability probing, and affords defenders opportunities to disinform attackers by divulging “fake” secrets in response to attempted intrusions. In order for such deceptions to succeed, honey-patched programs must be imbued with the ability to impersonate unpatched software with all secrets replaced by honey-data. That is, they require a technology for rapidly and thoroughly redacting all secrets from the victim program’s address space at runtime, yielding a vulnerable process that the attacker may further penetrate without risk of secret disclosure.

Realizing such runtime process secret redaction in practice elicits at least two significant challenges. First, the redaction step must yield a runnable program process. Non-secrets must therefore not be conservatively redacted, lest data critical for continuing the program’s execution be deleted. Secret redaction for running processes is hence especially sensitive to *label creep* and *over-tainting* failures. Second, many real-world programs targeted by cyber-attacks were not originally designed with information flow tracking support, and are often expressed in low-level, type-unsafe languages, such as C/C++. A suitable solution must be amenable to retrofitting such low-level, legacy software with annotations sufficient to distinguish non-secrets from secrets, and with efficient flow-tracking logic that does not impair performance.

[1] *Lattner and Adve (2004)*

[2] *DFSan (2016)*

Our approach builds upon the LLVM compiler’s[1] DataFlow Sanatizer (DFSan) infrastructure,[2] which adds byte-granularity taint-tracking support to C/C++ programs at compile-time. At the source level, DFSan’s taint-tracking capabilities are purveyed as runtime data-classification, data-declassification, and taint-checking operations, which programmers add to their programs to identify secrets and curtail their flow at runtime. Unfortunately, straightforward use of this interface for redaction of large, complex legacy codes can lead to severe over-tainting, or requires an unreasonably detailed retooling of the code with copious classification operations. This is unsafe, since missing even one of these classification points during retooling risks disclosing secrets to adversaries.

To overcome these deficiencies, we augment DFSan with a declarative, type annotation-based secret-labeling mechanism for easier secret identification; and we introduce a new label propagation semantics, called *Pointer Conditional-Combine Semantics* (PC^{2}S), that efficiently distinguishes secret data within C-style graph data structures from the non-secret structure that houses the data. This partitioning of the bytes greatly reduces over-tainting and the programmer's annotation burden, and proves critical for precisely redacting secret process data whilst preserving process operation after redaction.

Our innovations are showcased through the development of a taint tracking-based honey-patching framework for three production web servers, including the popular Apache HTTP server (~2.2M SLOC). The modified servers respond to detected intrusions by transparently forking attacker sessions to unpatched process clones in confined decoy environments. Runtime redaction preserves attacker session data without preserving data owned by other users, yielding a deceptive process that continues servicing the attacker without divulging secrets. The decoy can then monitor attacker strategies, harvest attack data, and disinform the attacker with honey-data in the form of false files or process data.

Our contributions can be summarized as follows:

- We introduce a pointer tainting methodology through which secret sources are derived from statically annotated data structures, lifting the burden of identifying classification code-points in legacy C code.

- We propose and formalize taint propagation semantics that accurately track secrets while controlling taint spread. Our solution is implemented as a small extension to LLVM, allowing it to be applied to a large class of COTS applications.

- We implement a memory redactor for secure honey-patching. Evaluation shows that our implementation is both more efficient and more secure than previous pattern-matching based redaction approaches.

- Implementations and evaluations for three production web servers demonstrate that the approach is feasible for large-scale, performance-critical software with reasonable overheads.

3.1 SOURCING AND TRACKING SECRETS

Taint-tracking conceptually entails labeling each byte of process memory with a security label that denotes its classification level. At compile-time, a taint-tracking compiler instruments the resulting object code with extra code that propagates these labels alongside the data they label. Extending such taint-tracking to low-level, legacy code not designed with taint-tracking in mind is often difficult. For example, the standard approach of specifying taint introductions as annotated program inputs often proves too coarse for inputs comprising low-level, unstructured data streams, such as network sockets. Listing 3.1 exemplifies the problem using a code excerpt from the Apache web server.[1] The excerpt partitions a byte stream (stored in buffer `s1`) into a non-secret user name and a secret password, delimited by a colon character. Naïvely labeling input `s1` as secret to secure the password causes the compiler to over-taint the user name (and the colon delimiter, and the rest of the stream), leading to excessive over-tainting—everything associated with the stream becomes secret, with the result that nothing can be safely divulged.

[1] *Apache (2014)*

A correct solution must more precisely identify data field `uptr→password` (but not `uptr→user`) as secret after the unstructured data has been parsed. This is achieved in DFSan by manually inserting a runtime classification operation after line 6. However, on a larger scale this brute-force labeling strategy imposes a dangerously heavy annotation burden on developers, who must manually locate all such classification points. In C/C++ programs littered with pointer arithmetic, the correct classification points can often be obscure. Inadvertently omitting even one classification risks information leaks.

Listing 3.1: Apache's URI parser function (excerpt)

```
/∗ first colon delimits username:password ∗/
s1 = memchr(hostinfo, ':', s − hostinfo);
if (s1) {
    uptr→user = apr_pstrmemdup(p, hostinfo, s1−hostinfo);
    ++s1;
    uptr→password = apr_pstrmemdup(p, s1, s − s1);
}
```

Listing 3.2: Apache's session record (excerpt)

```
typedef struct {
    NONSECRET apr_pool_t ∗pool;
    NONSECRET apr_uuid_t ∗uuid;
    SECRET_STR const char ∗remote_user;
    apr_table_t ∗entries;
    ...
} SECRET session_rec;
```

To ease this burden, a better solution is to introduce a mechanism whereby developers can identify secret-storing structures and fields *declaratively* rather than operationally. For example, to correctly label the password in Listing 3.1 as secret, users may add type qualifier SECRET_STR to the password field's declaration in its abstract datatype definition. A modified LLVM compiler responds to this static annotation by instrumenting the program with instructions that dynamically taint all values assigned to the password field. Since datatypes typically have a single point of definition (in contrast to the many code points that access them), this greatly reduces the annotation burden imposed upon code maintainers.

In cases where the appropriate taint is not statically known (e.g., if each password requires a different, user-specific taint label), parameterized type-qualifier SECRET⟨f⟩ identifies a user-implemented function f that computes the appropriate taint label at runtime.

Unlike traditional taint introduction semantics, which label program input values and sources with taints, recognizing structure fields as taint sources requires a new form of taint semantics that conceptually interprets dynamically identified *memory addresses* as taint sources. For example, a program that assigns address `&(uptr→password)` to pointer variable p, and then assigns a freshly allocated memory address to $*\mathsf{p}$, must automatically identify the freshly allocated memory as a new taint source, and thereafter taint any values stored at $*\mathsf{p}[\mathsf{i}]$ (for all indexes i).

To achieve this, DFSan's *pointer-combine semantics (PCS)* feature is extended to optionally combine (i.e., join) the taints of pointers and pointees during pointer dereferences. Specifically, when *PCS on-load* is enabled, read-operation $*\mathsf{p}$ yields a value tainted with the join of pointer p's taint and the taint of the value to which p points; and when *PCS on-store* is enabled, write-operation $*\mathsf{p} := \mathsf{e}$ taints the value stored into $*\mathsf{p}$ with the join of p's and e's taints. Using PCS leads to a natural encoding of `SECRET` annotations as pointer taints. Continuing the previous example, PCS propagates `uptr→password`'s taint to p, and subsequent dereferencing assignments propagate the two pointers' taints to secrets stored at their destinations.

PCS works well when secrets are always separated from the structures that house them by a level of pointer indirection, as in the example above (where `uptr→password` is a pointer to the secret rather than the secret itself). However, label creep difficulties arise when structures mix secret values with non-secret pointers. To illustrate, consider a singly linked list ℓ of secret integers, where each integer has a different taint. In order for PCS on-store to correctly classify values stored to ℓ`→secret_int`, pointer ℓ must have taint γ_1, where γ_1 is the desired taint of the first integer. But this causes stores to ℓ`→next` to incorrectly propagate taint γ_1 to the node's next-pointer, which propagates γ_1 to subsequent nodes when dereferenced. In the worst case, all nodes become labeled with all taints. Such issues have spotlighted effective pointer tainting as a significant challenge in the taint-tracking literature.[1]

[1] *Dalton et al. (2010); Slowinska and Bos (2009); Kang et al. (2011); Schwartz et al. (2010)*

To address this shortcoming, PC^{2}S semantics generalize PCS semantics by augmenting them with pointer-combine *exemptions* conditional upon the static type of the pointee. In particular, a PC^{2}S taint-propagation policy may dictate that taint labels are not combined when the pointee has pointer type. Hence, ℓ→`secret_int` receives ℓ's taint because the assigned expression has integer type, whereas ℓ's taint is *not* propagated to ℓ→`next` because the latter's assigned expression has pointer type. Empirical evaluation shows that just a few strategically selected exemption rules expressed using this refined semantics suffices to vastly reduce label creep while correctly tracking all secrets in large legacy source codes.

In order to strike an acceptable balance between security and usability, the solution only automates tainting of C/C++ style structures whose non-pointer fields share a common taint. Non-pointer fields of mixed taintedness within a single struct are not supported automatically because C programs routinely use pointer arithmetic to reference multiple fields in a struct via a common pointer (imparting the pointer's taint to all the struct's non-pointer fields)—for example, when copying structures, or when marshalling and demarshalling them to/from streams. This approach therefore targets the common case in which the taint policy is expressible at the granularity of structures, with exemptions for fields that point to other (differently tainted) structure instances. This corresponds to the usual scenario where a non-secret graph structure (e.g., a tree) stores secret data in its nodes.

With these new language extensions, users label structure datatypes as `SECRET` (implicitly introducing a taint to all fields within the structure), and additionally annotate pointer fields as `NONSECRET` to exempt their taints from pointer-combines during dereferences. Pointers to dynamic-length, null-terminated secrets get annotation `SECRET_STR`. For example, Listing 3.2 illustrates the annotation of `session_req`, used by Apache to store remote users' session data. Finer-

programs	$\mathcal{P} ::= \overline{c}$	*locations*	$\ell ::=$ memory addresses
commands	$c ::= v{:=}e \mid \texttt{store}(\tau, e_1, e_2)$	*environment*	$\Delta : v \rightharpoonup u$
	$\mid \texttt{ret}(\tau,\ e) \mid \texttt{br}(e, e_1, e_0)$	*prog counter*	pc
	$\mid \texttt{call}(\tau, e, \overline{\texttt{args}})$	*stores*	$\sigma : (\ell \rightharpoonup u) \cup (v \rightharpoonup \ell)$
expressions	$e ::= v \mid \langle u, \gamma \rangle \mid \diamond_b(\tau, e_1, e_2)$	*functions*	f
	$\mid \texttt{load}(\tau, e)$	*function table*	$\phi : f \rightharpoonup \ell$
binary ops	$\diamond_b ::=$ typical binary operators	*taint contexts*	$\lambda : (\ell \cup v) \rightharpoonup \gamma$
variables	v	*propagation*	$\rho : \overline{\gamma} \to \gamma$
values	$u ::=$ values of underlying IR	*prop contexts*	$\mathcal{A} : f \to \rho$
types	$\tau ::= ptr\ \tau \mid \tau\ \overline{\tau} \mid$ primitive types	*call stack*	$\Xi ::= nil$
taint labels	$\gamma \in (\Gamma, \sqsubseteq)$ (label lattice)		$\mid \langle f,\ pc,\ \Delta,\ \overline{\gamma} \rangle :: \Xi$

Figure 3.1: Intermediate representation syntax

granularity policies remain enforceable, but require manual instrumentation via DFSan's API, to precisely distinguish which of the code's pointer dereference operations propagate pointer taints. This solution thus complements existing approaches.

3.2 FORMAL SEMANTICS

For explanatory precision, the new taint-tracking semantics is formally defined in terms of the simple, typed intermediate language (IL) in Figure 3.1, inspired by prior work.[1] The simplified IL abstracts irrelevant details of LLVM's IR language, capturing only those features needed to formalize the analysis.

[1] *Schwartz et al. (2010)*

LANGUAGE SYNTAX Programs $\mathcal{P}$ are lists of commands, denoted $\overline{c}$. Commands consist of variable assignments, pointer-dereferencing assignments (stores), conditional branches, function invocations, and function returns. Expressions evaluate to value-taint pairs $\langle u, \gamma \rangle$, where u ranges over typical value representations, and γ is the taint label associated with u.

Labels denote sets of taints; they therefore comprise a lattice ordered by subset ($\sqsubseteq$), with the empty set $\bot$ at the bottom (denoting public data), and the universe $\top$ of all taints at the top (denoting maximally secret data). Join operation $\sqcup$ denotes least upper bound (union) of taint sets.

Variable names range over identifiers and function names, and the type system supports pointer types, function types, and typical primitive types. Since DFSan's taint-tracking is dynamic, we here omit a formal static semantics and assume that programs are well-typed. Execution contexts are comprised of a store σ relating locations to values and variables to locations, an environment Δ mapping variables to values, and a tainting context λ mapping locations and variables to taint labels. Additionally, to express the semantics of label propagation for external function calls (e.g., runtime library API calls), function table ϕ maps external function names to their entry points, a propagation context $\mathcal{A}$ that dictates whether and how each external function propagates its argument labels to its return value label, and the call stack Ξ. Taint propagation policies returned by $\mathcal{A}$ are expressed as customizable mappings ρ from argument labels $\overline{\gamma}$ to return labels γ.

OPERATIONAL SEMANTICS Figure 3.2 presents an operational semantics defining how taint labels propagate in an instrumented program. Expression judgments are large-step ($\Downarrow$), while command judgments are small-step ($\rightarrow_1$). At the IL level, expressions are pure and programs are non-reflective. Abstract machine configurations consist of tuples $\langle \sigma, \Delta, \lambda, \Xi, pc, \iota \rangle$, where pc is the program pointer and ι is the current instruction. Notation $\Delta[v \mapsto u]$ denotes function Δ with v remapped to u, and notation $\mathcal{P}[pc]$ refers to the program instruction at address pc. For brevity, we omit $\mathcal{P}$ from machine configurations, since it is fixed.

Rule VAL expresses the typical convention that hardcoded program constants are initially untainted ($\bot$). Binary operations are eager, and label their outputs with the join ($\sqcup$) of their operand labels. The semantics of `load`(τ, e) read the

$$\frac{}{\sigma, \Delta, \lambda \vdash u \Downarrow \langle u, \bot \rangle}\ \textsc{Val} \qquad \frac{}{\sigma, \Delta, \lambda \vdash v \Downarrow \langle \Delta(v), \lambda(v) \rangle}\ \textsc{Var}$$

$$\frac{\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1 \rangle \quad \sigma, \Delta, \lambda \vdash e_2 \Downarrow \langle u_2, \gamma_2 \rangle}{\sigma, \Delta, \lambda \vdash \diamond_b(\tau,\ e_1,\ e_2) \Downarrow \langle u_1\ \diamond_b\ u_2, \gamma_1 \sqcup \gamma_2 \rangle}\ \textsc{BinOp}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle}{\sigma, \Delta, \lambda \vdash \texttt{load}(\tau,\ e) \Downarrow \langle \sigma(u), \rho_{load}(\tau, \gamma, \lambda(u)) \rangle}\ \textsc{Load}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle \quad \Delta' = \Delta[v \mapsto u] \quad \lambda' = \lambda[v \mapsto \gamma]}{\langle \sigma, \Delta, \lambda, \Xi, pc, v := e \rangle \to_1 \langle \sigma, \Delta', \lambda', \Xi, pc + 1, \mathcal{P}[pc + 1] \rangle}\ \textsc{Assign}$$

$$\frac{\begin{array}{c}\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1 \rangle \\ \sigma, \Delta, \lambda \vdash e_2 \Downarrow \langle u_2, \gamma_2 \rangle \quad \sigma' = \sigma[u_1 \mapsto u_2] \quad \lambda' = \lambda[u_1 \mapsto \rho_{store}(\tau, \gamma_1, \gamma_2)]\end{array}}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{store}(\tau, e_1, e_2) \rangle \to_1 \langle \sigma', \Delta, \lambda', \Xi, pc + 1, \mathcal{P}[pc + 1] \rangle}\ \textsc{Store}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle \qquad \sigma, \Delta, \lambda \vdash e_{(u\,?\,1\,:\,0)} \Downarrow \langle u', \gamma' \rangle}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{br}(e, e_1, e_0) \rangle \to_1 \langle \sigma, \Delta, \lambda, \Xi, u', \mathcal{P}[u'] \rangle}\ \textsc{Cond}$$

$$\frac{\begin{array}{c}\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1 \rangle\ \cdots\ \sigma, \Delta, \lambda \vdash e_n \Downarrow \langle u_n, \gamma_n \rangle \quad \Delta' = \Delta[\overline{params_f} \mapsto \overline{u_1 \cdots u_n}] \\ \lambda' = \lambda[\overline{params_f} \mapsto \overline{\gamma_1 \cdots \gamma_n}] \quad fr = \langle f, pc + 1, \Delta, \overline{\gamma_1 \cdots \gamma_n} \rangle\end{array}}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{call}(\tau, f, \overline{e_1 \cdots e_n}) \rangle \to_1 \langle \sigma, \Delta', \lambda', fr :: \Xi, \phi(f), \mathcal{P}[\phi(f)] \rangle}\ \textsc{Call}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle \quad fr = \langle f, pc', \Delta', \overline{\gamma} \rangle \quad \lambda' = \lambda[v_{ret} \mapsto \mathcal{A}\, f\, \overline{\gamma}]}{\langle \sigma, \Delta, \lambda, fr :: \Xi, pc, \texttt{ret}(\tau, e) \rangle \to_1 \langle \sigma, \Delta'[v_{ret} \mapsto u], \lambda', \Xi, pc', \mathcal{P}[pc'] \rangle}\ \textsc{Ret}$$

Figure 3.2: Operational semantics of a generalized label propagation semantics

value stored in location e, where the label associated with the loaded value is obtained by propagation function ρ_{load}. Dually, $\texttt{store}(\tau, e_1, e_2)$ stores e_2 into location e_1, updating λ according to ρ_{store}. In C programs, these model pointer dereferences and dereferencing assignments, respectively. Parameterizing these rules in terms of abstract propagation functions ρ_{load} and ρ_{store} allows us to instantiate them with customized propagation policies at compile-time, as detailed in Section 3.2.

External function calls $\texttt{call}(\tau, f, \overline{e_1 \cdots e_n})$ evaluate arguments $\overline{e_1 \cdots e_n}$, create a new stack frame fr, and jump to the callee's entry point. Returns then consult propagation context $\mathcal{A}$ to appropriately label the value returned by the

NCS	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := \gamma_2$
PCS	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := \gamma_1 \sqcup \gamma_2$
PC²S	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := (\tau \text{ is } ptr)\ ?\ \gamma_2 : (\gamma_1 \sqcup \gamma_2)$

Figure 3.3: Polymorphic functions for no-combine, pointer-combine, and PC²S propagation policies

function based on the labels of its arguments. Context $\mathcal{A}$ can be customized by the user to specify how labels propagate through external libraries compiled without taint-tracking support.

LABEL PROPAGATION SEMANTICS The operational semantics are parameterized by propagation functions ρ that can be instantiated to a specific propagation policy at compile-time. This provides a base framework through which we can study different propagation policies and their differing characteristics. Figure 3.3 presents three polymorphic[1] functions that can be used to instantiate propagation policies. On-load propagation policies instantiate ρ_{load}, while on-store policies instantiate ρ_{store}. The instantiations in Figure 3.3 define no-combine semantics (DFSan's on-store default), PCS (DFSan's on-load default), and our PC²S extensions:

[1] The functions are polymorphic in the sense that some of their arguments are types τ.

NO-COMBINE The no-combine semantics (NCS) model a traditional, pointer-transparent propagation policy. Pointer labels are ignored during loads and stores, causing loaded and stored data retain their labels irrespective of the labels of the pointers being dereferenced.

POINTER-COMBINE SEMANTICS In contrast, PCS joins pointer labels with loaded and stored data labels during loads and stores. Using this policy, a value is tainted on-load (resp., on-store) if its source memory location (resp., source operand) is tainted or the pointer value dereferenced during the operation is tainted. If both are tainted with different

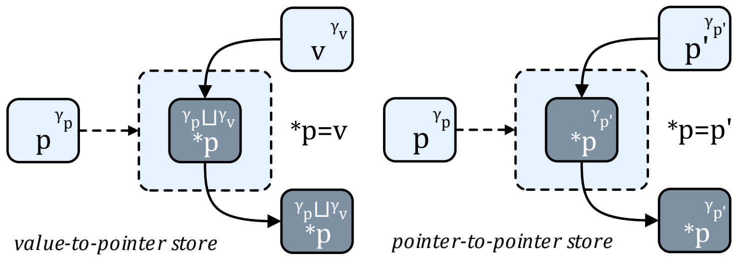


Figure 3.4: PC²S propagation policy on store commands

labels, the labels are joined to obtain a new label that denotes the union of the originals.

POINTER CONDITIONAL-COMBINE SEMANTICS PC²S generalizes PCS by conditioning the label-join on the static type of the data operand. If the loaded/stored data has pointer type, it applies the NCS rule; otherwise, it applies the PCS rule. The resulting label propagation for stores is depicted in Figure 3.4.

This can be leveraged to obtain the best of both worlds. PC²S pointer taints retain most of the advantages of PCS—they can identify and track aliases to birthplaces of secrets, such as data structures where secrets are stored immediately after parsing, and they automatically propagate their labels to data stored there. But PC²S resists PCS's over-tainting and label creep problems by avoiding propagation of pointer labels through levels of pointer indirection, which usually encode relationships with other data whose labels must remain distinct and separately managed.

Condition $(\tau \text{ is } \mathit{ptr})$ in Figure 3.3 can be further generalized to any decidable proposition on static types τ. This feature is used to distinguish pointers that cross data ownership boundaries (e.g., pointers to other instances of the parent structure) from pointers that target value data (e.g., strings). The former receive NCS treatment by default to resist over-tainting, while the latter receive PCS treatment by default to capture secrets and keep the annotation burden low. In addition, PC²S is at least as efficient as PCS because propagation policy ρ is partially evaluated at compile-time. Thus, the choice of NCS or PCS semantics for each pointer operation is decided purely statically, conditional upon the static types of the operands. The appropriate specialized propagation implementation is then in-lined into the resulting object code during compilation.

EXAMPLE To illustrate how each semantics propagates taint, consider the following IL pseudo-code, which revisits the linked-list example informally presented in Section 3.1.

```
store($\tau_{id}$, $request\_id$, parse($s$, $id\_size$));
store($\tau_{key}$, $p[request\_id]$→key, parse($s$, $key\_size$));
store($\tau_{ctx\_t*}$, $p[request\_id]$→next, $queue\_head$);
```

Input stream s includes a non-secret request identifier and a secret key of primitive type (e.g., unsigned long). If one labels stream s secret, then the public $request\_id$ becomes over-tainted in all three semantics, which is undesirable because a redaction of $request\_id$ may crash the program (when $request\_id$ is later used as an array index). A better solution is to label pointer p secret and employ PCS, which correctly labels the key at the moment it is stored. However, PCS additionally taints the next-pointer, leading to over-tainting of all the nodes in the containing linked-list, some of which may contain keys owned by other users. PC^{2}S avoids this over-tainting by exempting the next pointer from the combine-semantics. This preserves the data structure while correctly labeling the secret data it contains.

3.3 AN INTEGRATED SECRET-REDACTING, HONEY-PATCHING ARCHITECTURE

Figure 3.5 presents the architecture of SignaC[1] (Secret Information Graph iNstrumentation for Annotated C),[2] which leverages compiler-instrumented secret-redaction to achieve

[1] *named after* pointillism *co-founder Paul Signac*

[2] *Araujo and Hamlen (2015)*

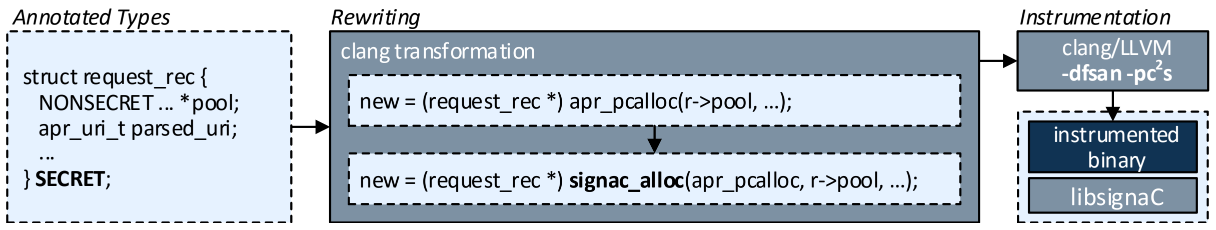


Figure 3.5: Architectural overview of SignaC illustrating its three-step, static instrumentation process: (1) annotation of security-relevant types, (2) source-code rewriting, and (3) compilation with the sanitizer's instrumentation pass

secret-sanitized process migration for secure honey-patching. At a high level, it consists of three components: (1) a source-to-source preprocessor, which (a) automatically propagates user-supplied, source-level type annotations to containing datatypes, and (b) in-lines taint introduction logic into dynamic memory allocation operations; (2) a modified LLVM compiler that instruments programs with PC^{2}S taint propagation logic during compilation; and (3) a runtime library that the instrumented code invokes during program execution to introduce taints and perform redaction.

3.3.1 *Source-code Rewriting*

TYPE ATTRIBUTES Server code maintainers first annotate data structures containing secrets with the type qualifier `SECRET`. This instructs the taint-tracker to treat all instantiations (e.g., dynamic allocations) of these structures as taint sources. Additionally, qualifier `NONSECRET` may be applied to pointer fields within these structures to exempt them from PCS. The instrumentation pass generates NCS logic instead for operations involving such members. Finally, qualifier `SECRET_STR` may be applied to pointer fields whose destinations are dynamic-length byte sequences bounded by a null terminator (strings).

To avoid augmenting the source language's grammar, these type qualifiers are defined using source-level attributes (specified with `__attribute__`) followed by a specifier. `SECRET` uses the `annotate` specifier, which defines a purely syntactic qualifier visible only at the compiler's front-end. In contrast, `NONSECRET` and `SECRET_STR` are required during the back-end instrumentation. To this end, we leverage Quala,[1] which extends LLVM with an overlay type system. Quala's `type_annotate` specifier propagates the type qualifiers throughout the IL code.

[1] *Sampson (2014)*

TYPE ATTRIBUTE REWRITING In the preprocessing step, the target application undergoes a source-to-source transformation pass that rewrites all dynamic allocations of annotated data types with taint-introducing wrappers. Implementing this transformation at the source level allows us to utilize the full type information that is available at the compiler's front-end, including purely syntactic attributes such as `SECRET` annotations. The implementation leverages Clang's tooling API[2] to traverse and apply the desired transformations directly into the program's AST. At a high-level, the rewriting algorithm takes the following steps:

[2] *Clang (2016)*

1. It first amasses a list of all *security-relevant datatypes*, which are defined as (a) all structs and unions annotated `SECRET`, (b) all types defined as aliases (e.g., via typedef) of security-relevant datatypes, and (c) all structs and unions containing secret-relevant datatypes not separated from the containing structure by a level of pointer indirection (e.g., nested struct definitions). This definition is recursive, so the list is computed iteratively from the transitive closure of the graph of datatype definition references.

2. It next finds all calls to memory allocation functions (e.g., malloc, calloc) whose return values are *explicitly* or *implicitly* cast to a security-relevant datatype. Such calls are wrapped in calls to SignaC's runtime library, which dynamically introduces an appropriate taint label to the newly allocated structure.

The task of identifying memory allocation functions is facilitated by a user-supplied list that specifies the memory allocation API. This allows the rewriter to handle programs that employ custom memory management. For example, Apache defines custom allocators in its Apache Portable Runtime (APR) memory management interface.

3.3.2 *PC²S Instrumentation*

The instrumentation pass next introduces LLVM IR code during compilation that propagates taint labels during program execution. Our implementation extends DFSan with the PC²S label propagation policy specified in Section 3.2.

TAINT REPRESENTATION To support a large number of taint labels, DFSan adopts a low-overhead representation of labels as 16-bit integers, with new labels allocated sequentially from a pool. Rather than reserving 2^n labels to represent the full power set of a set of n primitive taints, DFSan lazily reserves labels denoting non-singleton sets on-demand. When a label union operation is requested at a join point (e.g., during binary operations on tainted operands), the instrumentation first checks whether a new label is required. If a label denoting the union has already been reserved, or if one operand label subsumes the other, DFSan returns the already-reserved label; otherwise, it reserves a fresh union label from the label pool. The fresh label is defined by pointers to the two labels that were joined to form it. Union labels are thus organized as a dynamically growing binary DAG—the *union table*.

This strategy benefits applications whose label-joins are sparse, visiting only a small subset of the universe of possible labels. Our PC²S semantics' curtailment of label creep thus synergizes with DFSan's lazy label allocation strategy, allowing us to realize taint-tracking for legacy code that otherwise exceeds the maximum label limit. This benefit is further evidenced in our evaluation (Section 3.4).

Table 3.1 shows the memory layout of an instrumented program. DFSan maps (without reserving) the lower 32 TB of the process address space for *shadow memory*, which stores

Table 3.1: Memory layout of an instrumented program

Start	End	Memory Region
0x700000008000	0x800000000000	application memory
0x200000000000	0x200200000000	union table
0x000000010000	0x200000000000	shadow memory
0x000000000000	0x000000010000	reserved by kernel

the taint labels of the values stored at the corresponding application memory addresses. This layout allows for efficient lookup of shadow addresses by masking and shifting the application's addresses. Labels of values not stored in memory (e.g., those stored in machine registers or optimized away at compile-time) are tracked at the IL level in SSA registers, and compiled to suitable taint-tracking object code.

FUNCTION CALLS Propagation context $\mathcal{A}$ defined in Section 3.2 models label propagation across external library function calls, expressed in DFSan as an Application Binary Interface (ABI). The ABI lists functions whose label-propagation behavior (if any) should be replaced with a fixed, user-defined propagation policy at call sites. For each such function, the ABI specifies how the labels of its arguments relate to the label of its return value.

DFSan natively supports three such semantics: (1) *discard*, which corresponds to propagation function $\rho_{dis}(\overline{\gamma}) := \bot$ (return value is unlabeled); (2) *functional*, corresponding to propagation function $\rho_{fun}(\overline{\gamma}) := \bigsqcup \overline{\gamma}$ (label of return value is the union of labels of the function arguments); and (3) *custom*, denoting a custom-defined label propagation wrapper function.

DFSan pre-defines an ABI list that covers glibc's interface. Users may extend this with the API functions of external libraries for which source code is not available or cannot be instrumented. For example, to instrument Apache with `mod_ssl`, we mapped OpenSSL's API functions to the ABI list. In addition, we extended the custom ABI wrappers of *memory transfer functions* (e.g., `strcpy`, `strdup`) and *input functions* (e.g., `read`, `pread`) to implement PC²S. For instance, we modified the wrapper for `strcpy(`*dest*`,`*src*`)` to taint *dest* with $\gamma_{src} \sqcup \gamma_{dest}$ when instrumenting code under PC²S.

STATIC INSTRUMENTATION The instrumentation pass is placed at the end of LLVM's optimization pipeline. This ensures that only memory accesses surviving all compiler optimizations are instrumented, and that instrumentation takes place just before target code is generated. Like other LLVM *transform* passes, the program transformation operates on LLVM IR, traversing the entire program to insert label propagation code. At the front-end, compilation flags parametrize the label propagation policies for the store and load operations discussed in Section 3.2.

STRING HANDLING Strings in C are not first-class types; they are implemented as character pointers. C's type system does not track their lengths or enforce proper termination. This means that purely static typing information is insufficient for the instrumentation to reliably identify strings or propagate their taints to all constituent bytes on store. To overcome this problem, users must annotate secret-containing, string fields with `SECRET_STR`. This cues the runtime library to taint up to and including the pointee's null terminator when a string is assigned to such a field. For safety, our runtime library (see Section 3.3.3) zeros the first byte of all fresh memory allocations, so that uninitialized strings are always null-terminated.

STORE INSTRUCTIONS Listing 3.3 summarizes the instrumentation procedure for stores in diff style. By default, DFSan instruments NCS on store instructions: it reads the shadow memory of the value operand (line 1) and copies it onto the shadow of the pointer operand (line 10). If PC²S is enabled (lines 2 and 11), the instrumentation consults the static type of the value operand and checks whether it is a non-pointer or non-exempt pointer field (which also subsumes

Listing 3.3: Store instruction instrumentation

```
 1    Value* Shadow = DFSF.getShadow(SI.getValueOperand());
 2  + if (Cl_PC2S_OnStore) {
 3  +     Type *t = SI.getValueOperand()→getType();
 4  +     if (!t→isPointerTy() || !isExemptPtr(&SI)) {
 5  +         Value *PtrShadow = DFSF.getShadow(SI.getPointerOperand());
 6  +         Shadow = DFSF.combineShadows(Shadow, PtrShadow, &SI);
 7  +     }
 8  + }
 9    ...
10    DFSF.storeShadow(SI.getPointerOperand(), Size, Align, Shadow, &SI);
11  + if (Cl_PC2S_OnStore) {
12  +     if (isSecretStr(&SI)) {
13  +         Value *Str = IRB.CreateBitCast(v, Type::getInt8PtrTy(Ctx));
14  +         IRB.CreateCall2(DFSF.DFS.DFSanSetLabelStrFn, Shadow, Str);
15  +     }
16  + }
```

SECRET_STR) in lines 3–4. If so, the shadows of the pointer and value operands are joined (lines 5–6), and the resulting label is stored into the shadow of the pointer operand. If the instruction stores a string annotated with SECRET_STR, the instrumentation calls a runtime library function that copies the computed shadow to all bytes of the null-terminated string (lines 12–15).

LOAD INSTRUCTIONS Listing 3.4 summarizes the analogous instrumentation for load instructions. First, the instrumentation loads the shadow of the value pointed by the pointer operand (line 1). If PC^{2}S is enabled (line 2), then the instrumentation checks whether the dereferenced pointer is tainted (line 3). If so, the shadow of the pointer operand is joined with the shadow of its value (lines 4–5), and the resulting label is saved to the shadow (line 9).

Listing 3.4: Load instruction instrumentation

```
  Value *Shadow = DFSF.loadShadow(LI.getPointerOperand(), Size, ...);
+ if (Cl_PC2S_OnLoad) {
+     if (!isExemptPtr(&LI)) {
+         Value *PtrShadow = DFSF.getShadow(LI.getPointerOperand());
+         Shadow = DFSF.combineShadows(Shadow, PtrShadow, &LI);
+     }
+ }
  ...
  DFSF.setShadow(&LI, Shadow);
```

Listing 3.5: Memory transfer intrinsics instrumentation

```
1 + if (Cl_PC2S_OnStore && !isExemptPtr(&I)) {
2 +     Value *DestShadow = DFSF.getShadow(I.getDest());
3 +     Value *SrcShadow = DFSF.getShadow(I.getSource());
4 +     DestShadow = DFSF.combineShadows(SrcShadow, DestShadow, &I);
5 +     DFSF.storeShadow(I.getDest(), Size, Align, DestShadow, &I);
6 + }
```

MEMORY TRANSFER INTRINSICS LLVM defines intrinsics for standard memory transfer operations, such as `memcpy` and `memmove`. These functions accept a source pointer *src*, a destination pointer *dst*, and the number of bytes *len* to be transferred. DFSan's default instrumentation destructively copies the shadow associated with *src* to the shadow of *dst*, which is not the intended propagation policy of PC^{2}S. We therefore instrument these functions as shown in Listing 3.5. The instrumentation reads the shadows of *src* and *dst* (lines 2–3), computes the union of the two shadows (line 4), and stores the combined shadows to the shadow of *dst* (line 5).

3.3.3 *Runtime Library*

The source-to-source rewriter and instrumentation phases in-line logic that calls a tiny dedicated library at runtime

Listing 3.6: Taint-introducing memory allocations

```
1 #define signac_alloc(alloc, args...)  ({ \
2       void *_p = alloc ( args ); \
3       signac_taint(&_p, sizeof(void*)); \
4       _p; })
```

to introduce taints, handle special taint-propagation cases (e.g., string support), and check taints at sinks (e.g., during redaction). The library exposes three API functions:

- `signac_init(`*pl*`)`: initialize a tainting context with a fresh label instantiation *pl* for the current principal.
- `signac_taint(`*addr*`,`*size*`)`: taint each address in interval $[addr, addr{+}size)$ with *pl*.
- `signac_alloc(`*alloc*`,...)`: wrap allocator *alloc* and taint the address of its returned pointer with *pl*.

Function `signac_init` instantiates a fresh taint label and stores it in a thread-global context, which function f of annotation SECRET⟨f⟩ may consult to identify the owning principal at taint-introduction points. In typical web server architectures, this function is strategically hooked at the start of a new connection's processing cycle. Function `signac_taint` sets the labels of each address in interval $[addr, addr{+}size)$ with the label *pl* retrieved from the session's context.

Listing 3.6 details `signac_alloc`, which wraps allocations of SECRET-annotated data structures. This variadic macro takes a memory allocation function *alloc* and its arguments, invokes it (line 2), and taints the address of the pointer returned by the allocator (line 3).

3.3.4 *Example: Apache Instrumentation*

To instrument a particular server application, such as Apache, SignaC requires two small, one-time developer interventions:

First, add a call to `signac_init` at the start of a user session to initialize a new tainting context for the newly identified principal. Second, annotate the security-relevant data structures whose instances are to be tracked. For instance, in Apache, `signac_init` is called upon the acceptance of a new server connection, and annotated types include `request_rec`, `connection_rec`, `session_rec`, and `modssl_ctx_t`. These are the structures where Apache stores URI parameters and request content information, private connection data such as remote IPs, key-value entries in user sessions, and encrypted connection information. The redaction scheme instruments the server with PC^{2}S. At redaction time, it scans the resulting shadow memory for labels denoting secrets owned by user sessions other than the attacker's, and redacts such secrets. The shadow memory and taint-tracking libraries are then unloaded, leaving a decoy process that masquerades as undefended and vulnerable.

3.4 EMPIRICAL EVALUATION

This section demonstrates the practical advantages and feasibility of our approach for retrofitting large legacy C codes with taint-tracking, through the development and evaluation of a honey-patching memory redaction architecture for three production web servers. All experiments were performed on a quad-core VM with 8 GB RAM running 64-bit Ubuntu 14.04. The host machine is an Intel Xeon E5645 workstation running 64-bit Windows 7.

3.4.1 *Honey-patching*

Figure 3.6 illustrates how honey-patches respond to intrusions by cloning attacker sessions to decoys. Upon intrusion detection, the honey-patch forks a shallow, local clone of the victim process. The cloning step redacts all secrets from the clone's address space, optionally replacing them with

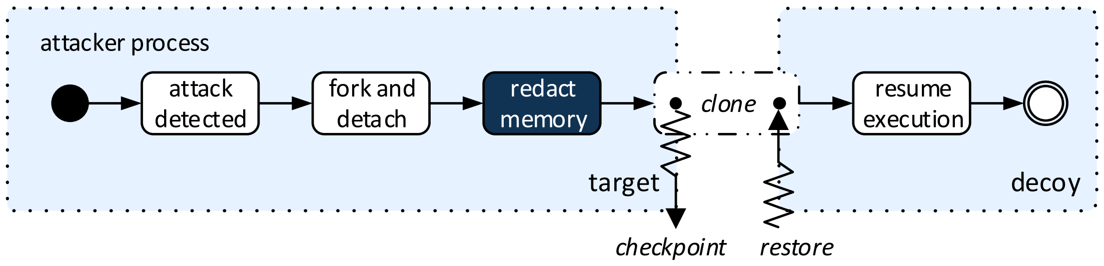


Figure 3.6: Honey-patch response to an intrusion attempt

honey-data. It then resumes execution in the decoy by emulating an unpatched implementation. This impersonates a successful intrusion, luring the attacker away from vulnerable victims, and offering defenders opportunities to monitor and disinform adversaries.

Chapter 2 implements secret redaction as a brute-force memory sweep that identifies and replaces plaintext string secrets. This is both slow and potentially unsafe; the sweep constitutes a majority of the response delay overhead during cloning,[1] and it can miss binary data secrets difficult to express reliably as regular expressions. Using SIGNAC, we implemented an information flow-based redaction strategy for honey-patching that is faster and more reliable than prior approaches.

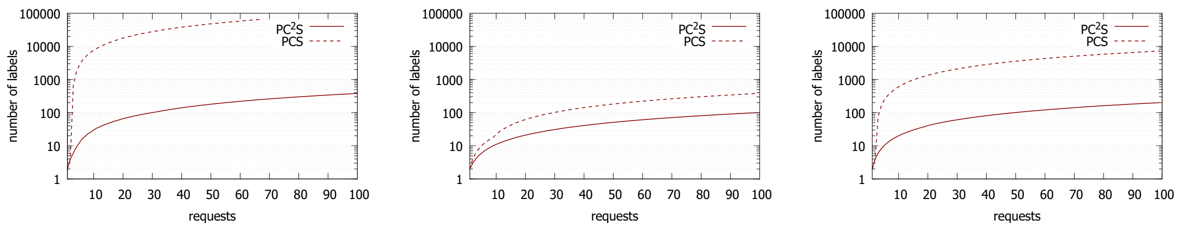
[1] *Araujo et al. (2014)*

Our redaction scheme instruments the server with dynamic taint-tracking. At redaction time, it scans the resulting shadow memory for labels denoting secrets owned by user sessions other than the attacker's, and redacts such secrets. The shadow memory and taint-tracking libraries are then unloaded, leaving a decoy process that masquerades as undefended and vulnerable.

EVALUATED SOFTWARE We implemented taint tracking-based honey-patching for three production web servers: Apache, Nginx, and Lighttpd. Apache and Nginx are the top two servers of all active websites, with 50.1% and 14.8% market share, respectively.[2] Apache comprises 2.27M SLOC mostly in C.[3] Nginx and Lighttpd are smaller, having about 146K and

[2] *Netcraft (2015)*

[3] *Ohloh (2014)*



(a) Apache (b) Nginx (c) Lighttpd

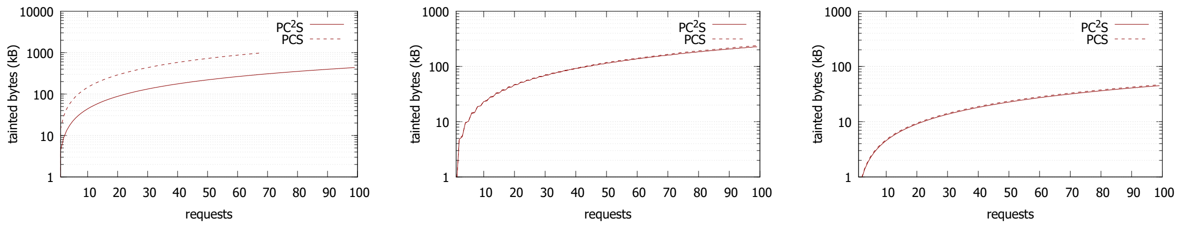
Figure 3.7: Experiment comparing label creeping behavior of PC^{2}S and PCS on Apache, Nginx, and Lighttpd

138K SLOC, respectively. All three are commercial-grade, feature-rich, open-source software products without any built-in support for information flow tracking.

To augment these products with PC^{2}S-style taint-tracking support, we manually annotated secret-storing structures and pointer fields. Altogether, we added approximately 45, 30, and 25 such annotations to Apache, Nginx, and Lighttpd, respectively. For consistent evaluation comparisons, we only annotated Apache's core modules for serving static and dynamic content, encrypting connections, and storing session data; we omitted its optional modules. We also manually added about 20–30 SLOC to each server to initialize the taint-tracker. Considering the sizes and complexity of these products, we consider the PC^{2}S annotation burden exceptionally light relative to prior approaches.

3.4.2 *Taint Spread*

OVER-TAINTING PROTECTION To test our approach's resistance to taint explosions, we submitted a stream of (non keep-alive) requests to each instrumented web server, recording a cumulative tally of distinct labels instantiated during taint-tracking. Figure 3.7 plots the results, comparing traditional PCS to our PC^{2}S extensions. On Apache, traditional PCS is impractical, exceeding the maximum label limit in just 68 requests. In contrast, PC^{2}S instantiates vastly fewer labels



(a) Apache (b) Nginx (c) Lighttpd

Figure 3.8: Experiment comparing the effects of PC²S and PCS on the cumulative tally of bytes tainted on Apache, Nginx, and Lighttpd

(note that the y-axes are *logarithmic scale*). After extrapolation, we conclude that an average 16,384 requests are required to exceed the label limit under PC²S—well above the standard 10K-request TTL limit for worker threads.

Taint spread control is equally critical for preserving program functionality after redaction. To demonstrate, we repeated the experiment with a simulated intrusion after $n \in [1, 100]$ legitimate requests. Figure 3.8 plots the cumulative tally of how many bytes received a taint during the history of the run for each tested web server. While we observed pronounced label creep for the three web servers, our experiments reveal that Apache is much more prone to taint spread failures. Apache has a ubiquitous memory pool structure that is referenced, directly or indirectly, by all of its structures, leading PCS to exhibit excessive taint spread.

Unlike Apache, Nginx and Lighttpd are event-driven and single-threaded (one thread per worker process), and their request handling is based on a state model of the connection (i.e., a state machine with different processing phases triggered by connection events). These servers are much smaller, offering fewer features than Apache. Their data structures are also leaner, and have fewer levels of pointer indirection. In Lighttp, for example, almost everything is stored in a single `malloc` call placed at the beginning of the request handling cycle. Taint spread therefore tends to stay contained within

Table 3.2: Honey-patched security vulnerabilities

Software	Version	CVE-ID	Description
Bash*	4.3	CVE-2014-6271	Improper parsing of environment variables
OpenSSL*	1.0.1f	CVE-2014-0160	Buffer over-read in heartbeat protocol extension
Apache	2.2.21	CVE-2011-3368	Improper URL validation
Apache	2.2.9	CVE-2010-2791	Improper timeouts of keep-alive connections
Apache	2.2.15	CVE-2010-1452	Bad request handling
Apache	2.2.11	CVE-2009-1890	Request content length out of bounds
Apache	2.0.55	CVE-2005-3357	Bad SSL protocol check

*tested with Apache 2.4.6

the connection pool of these servers, and is almost identical despite the chosen taint propagation semantics.

In all cases, redaction crashed PCS-instrumented processes cloned after just 2–3 legitimate requests (due to erasure of over-tainted bytes). In contrast, PC^{2}S-instrumented processes never crashed; their decoy clones continued running after redaction, impersonating vulnerable servers. This demonstrates our approach's facility to realize effective taint-tracking in legacy codes for which prior approaches fail.

UNDER-TAINTING PROTECTION To double-check that PC^{2}S redaction was actually erasing all secrets, we created a workload of legitimate post requests with pre-seeded secrets to a web-form application. We then automated exploits of the honey-patched vulnerabilities listed in Table 3.2, including the famous Shellshock and Heartbleed vulnerabilities. For each exploit, we ran the brute-force memory sweep redactor developed in Chapter 2 after SignaC's redactor to confirm that the former finds no secrets missed by the latter. We also manually inspected memory dumps of each clone to confirm that none of the pre-seeded secrets survived. In all cases, the honey-patch responds to the exploits as a vulnerable decoy server devoid of secrets.

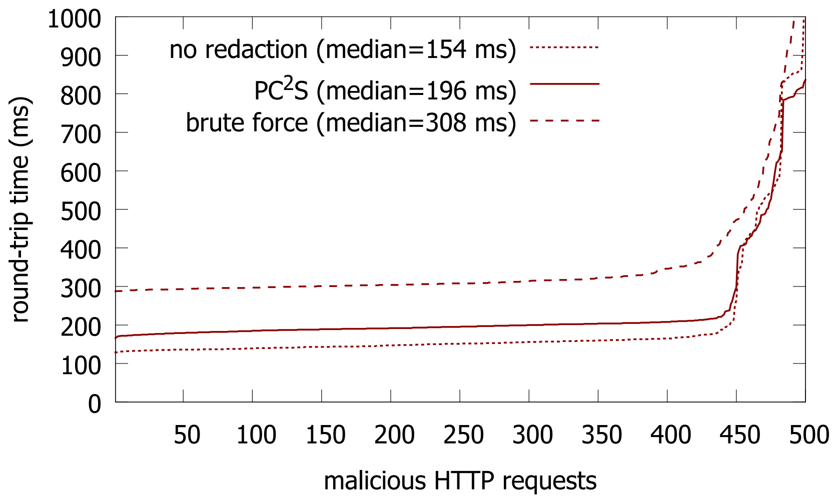


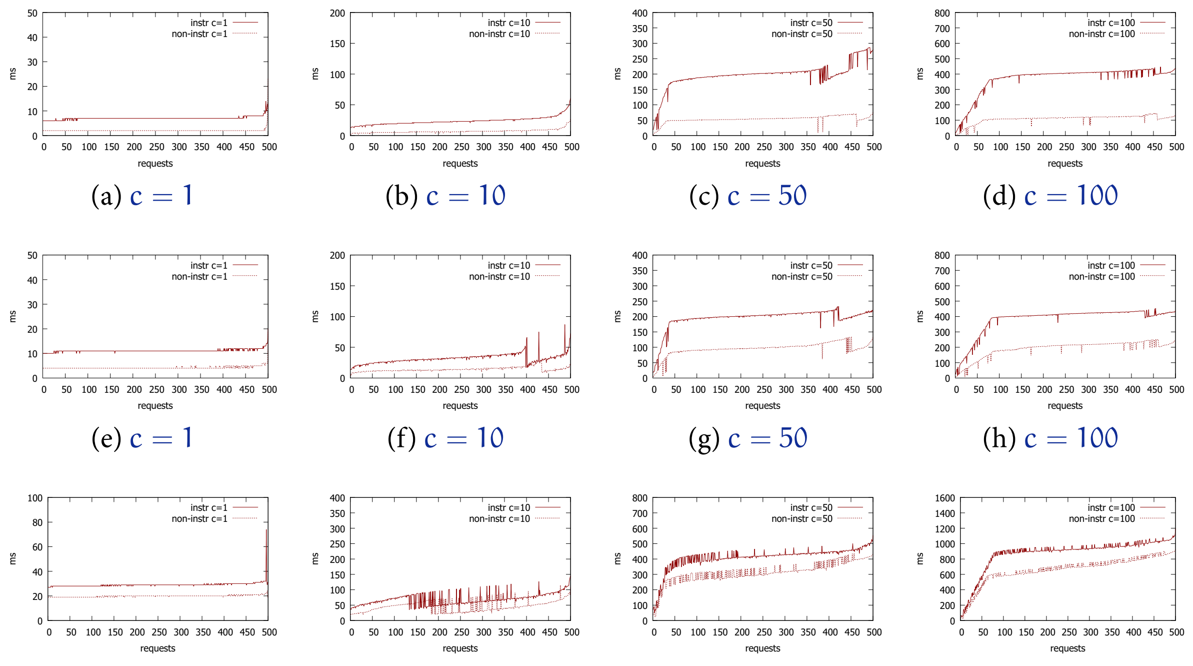
Figure 3.9: Request round-trip times for attacker session forking on honey-patched Apache

3.4.3 *Performance*

REDACTION PERFORMANCE To evaluate the performance overhead of redacting secrets, we benchmarked three honey-patched Apache deployments: (1) a baseline instance without memory redaction, (2) the brute-force memory sweep redactor from Chapter 2, and (3) our PC^{2}S redactor. We used Apache's server benchmarking tool (*ab*) to launch 500 malicious HTTP requests against each setup, each configured with a pool of 25 decoys.

Figure 3.9 shows request round-trip times for each deployment. PC^{2}S redaction is about 1.6$\times$ faster than brute-force memory sweep redaction; the former's request times average 0.196s, while the latter's average 0.308s. This significant reduction in cloning delay considerably improves the technique's deceptiveness, making it more transparent to attackers. Nginx and Lighttpd also exhibit improved response times of 16% (0.165s down to 0.138s) and 21% (0.155s down to 0.122s), respectively.

TAINT-TRACKING PERFORMANCE To evaluate the performance overhead of the static instrumentation, three



(a) $c = 1$ (b) $c = 10$ (c) $c = 50$ (d) $c = 100$

(e) $c = 1$ (f) $c = 10$ (g) $c = 50$ (h) $c = 100$

(i) $c = 1$ (j) $c = 10$ (k) $c = 50$ (l) $c = 100$

Figure 3.10: Dynamic taint-tracking performance (measured in request round-trip times) with varying concurrency c for a static web site (a–d), Bash CGI application (e–h), and PHP application (i–l)

Apache setups were tested: a static-content HTML website (~20 KB page size), a CGI-based Bash application that returns the server's environment variables, and a dynamic PHP website displaying the server's configuration. For each web server setup, *ab* was executed with four concurrency levels c (i.e., the number of parallel threads). Each run comprises 500 concurrent requests, plotted in ascendant order of their round-trip times (RTT).

Figure 3.10 shows the results for $c = 1, 10, 50$, and 100, and the average overheads observed for each test profile are summarized in Table 3.3. Our measurements show overheads of $2.4\times$, $1.1\times$, and $0.3\times$ for the static-content, CGI, and PHP websites, respectively, which is consistent with dynamic taint-tracking overheads reported in the prior literature.[1] Since server computation accounts for only about 10% of

[1] *Serebryany et al. (2012)*

Table 3.3: Average overhead of instrumentation

Benchmark	$c = 1$	$c = 10$	$c = 50$	$c = 100$
Static	2.50	2.34	2.56	2.32
CGI Bash	1.29	0.98	1.00	0.97
PHP	0.41	0.37	0.30	0.31

overall web site response delay in practice,[1] this corresponds to observable overheads of about 24%, 11%, and 3% (respectively).

[1] *Souders (2007)*

While such overhead characterizes feasibility, it is irrelevant to deception because unpatched, patched, and honey-patched vulnerabilities are all slowed equally by the taint-tracking instrumentation. The overhead therefore does not reveal which apparent vulnerabilities in a given server instance are genuine patching lapses and which are deceptions, and it does not distinguish honey-patched servers from servers that are slowed by any number of other factors (e.g., fewer computational resources). In addition, it is encouraging that high relative overheads were observed primarily for static websites that perform little or no significant computation. This suggests that the more modest 3% overhead for computationally heavier PHP sites is more representative of servers for which computational performance is an issue.

3.5 DISCUSSION

3.5.1 *Approach Limitations*

Our research significantly eases the task of tracking secrets within standard, pointer-linked, graph data-structures as they are typically implemented in low-level languages, like C/C++. However, there are many non-standard, low-level programming paradigms that our approach does not fully support automatically. Such limitations are discussed below.

POINTER PRE-ALIASES PC²S fully tracks all pointer aliases via taint propagation starting from the point of taint-introduction (e.g., the code point where a secret is first assigned to an annotated structure field after parsing). However, if the taint-introduction policy misidentifies secret sources too late in the program flow, dynamic tracking cannot track pointer *pre-aliases*—aliases that predate the taint-introduction. For example, if a program first initializes string p_1, then aliases $p_2 := p_1$, and finally initializes secret-annotated field f via $f := p_1$, PC²S automatically labels p_1 (and f) but not pre-alias p_2.

In most cases this mislabeling of pre-aliases can be mitigated by enabling PC²S both on-load and on-store. This causes secrets stored via p_2 to receive the correct label on-load when they are later read via p_1 or f. Likewise, secrets read via p_2 retain the correct label if they were earlier stored via p_1 or f. Thus, only data stored *and* read purely using independent pre-alias p_2 remain untainted. This is a correct enforcement of the user's policy, since the policy identifies $f := p_1$ as the taint source, not p_2. If this treatment is not desired, the user must therefore specify a more precise policy that identifies the earlier origin of p_1 as the true taint source (e.g., by manually inserting a dynamic classification operation where p_1 is born), rather than identifying f as the taint source.

STRUCTURE GRANULARITY Our automation of taint-tracking for graph data-structures implemented in low-level languages leads to taint annotations at the granularity of whole struct declarations, not individual value fields. Thus, all non-pointer fields within a secret-annotated C struct receive a common taint under our semantics. This coarse granularity is appropriate for C programs since such programs can (and often do) refer to multiple data fields within a given struct instance using a common pointer. For example, *marshalling* is typically implemented as a pointer-walk that reads a byte stream directly into all data fields (but not the pointer fields) of a struct instance byte-by-byte. All data fields therefore receive a common label after marshalling.

Reliable support for structs containing secrets of mixed taint therefore requires a finer-grained taint-introduction policy than is expressible by declarative annotations of C structure definitions. Such policies must be operationally specified in C through runtime classifications at secret-introducing code points. Our focus in this research is on automating the much more common case where each node of the graph structure holds secrets of uniform classification, toward lifting the user's annotation burden for this most common case.

DYNAMIC-LENGTH SECRETS Our implementation provides built-in support for a particularly common form of dynamic-length secret—null-terminated strings. This can be extended to support other forms of dynamic-length secrets as needed. For example, strings with an explicit length count rather than a terminator, fat and bounded pointers,[1] and other variable-length, dynamically allocated, data structures can be supported through the addition of an appropriate annotation type and a dynamic taint-propagating function that extends pointer taints to the entire pointee during assignments.

[1] *Jim et al. (2002)*

IMPLICIT FLOWS Our dynamic taint-tracking tracks explicit information flows, but not implicit flows that disclose information through control-flows rather than dataflows. Tracking implicit flows generally requires static information flow analysis to reason about disclosures through inaction (non-observed control-flows) rather than merely actions. Such analysis is often intractable (and generally undecidable) for low-level languages like C, whose control-flows include unstructured and dynamically computed transitions.

Likewise, dynamic taint-tracking does not monitor side-channels, such as resource consumption (e.g., memory or power consumption), runtimes, or program termination, which can also divulge information. For our problem domain (program process redaction), such channels are largely irrelevant, since attackers may only exfiltrate information after redaction, which leaves no secrets for the attacker to glean, directly or indirectly.

3.5.2 *Process Memory Redaction*

Our research introduces live process memory image sanitization as a new problem domain for information flow analysis. Process memory redaction raises unique challenges relative to prior information flow applications. It is exceptionally sensitive to over-tainting and label creep, since it must preserve process execution (e.g., for process debugging, continued service availability, or attacker deception); it demands exceptionally high performance; and its security applications prominently involve large, low-level, legacy codes, which are the most frequent victims of cyber-attacks. Future work should expand the search for solutions to this difficult problem to consider the suitability of other information flow technologies, such as static type-based analyses.[1]

[1] *cf., Sabelfeld and Myers (2003)*

3.5.3 *Language Compatibility*

While our implementation targets one particularly ubiquitous source language (C/C++), our general approach is applicable to other similarly low-level languages, as well as scripting languages whose interpreters are implemented in C (e.g., PHP, Bash). Such languages are common choices for implementing web services, and targeting them is therefore a natural next step for the web security thrust of our research.

3.6 CONCLUSION

PC^{2}S significantly improves the feasibility of dynamic taint-tracking for low-level legacy code that stores secrets in graph data structures. To ease the programmer's annotation burden and avoid taint explosions suffered by prior approaches, it introduces a novel pointer-combine semantics that resists taint over-propagation through graph edges. Our LLVM implementation extends C/C++ with declarative type qualifiers for secrets, and instruments programs with taint-tracking capabilities at compile-time.

The new infrastructure is applied to realize efficient, precise honey-patching of production web servers for attacker deception. The deceptive servers self-redact their address spaces in response to intrusions, affording defenders a new tool for attacker monitoring and disinformation.

4 DECEPTION-ENHANCED ANOMALY DETECTION

Detecting previously unseen cyber attacks before they reach unpatched, vulnerable computer systems (or afterward, for recovery purposes) has become a vital necessity for many organizations. In 2015 alone, a new zero-day vulnerability was found *every week*—more than doubling the previous year's rate—and over 75% of all legitimate websites have unpatched vulnerabilities, 20% of which afford attackers full control over victim systems.[1] The cost of data breaches resulting from software exploits is expected to escalate to a staggering $2.1 trillion by 2019.[2]

The research presented in this chapter was performed in collaboration with K. Al-Naami, A. Gbadebo, L. Khan, and K.W. Hamlen at the University of Texas at Dallas. This work is currently submitted for publication. Lead author Araujo led the honey-patching half of the research, including experimental design, data generation, and honey-patching implementation.

[1] *Symantec (2016)*

[2] *Juniper Research (2015)*

Intrusion detection[3] has long been championed as a means of mitigating such threats. The approach capitalizes on the observation that the most damaging and pernicious attacks discovered in the wild often share similar traits, such as the steps intruders take to open back doors, execute files and commands, alter system configurations, and transmit gathered information from compromised machines.[4] Starting with the initial infection, such malicious activities often leave telltale traces that can be identified even when the underlying exploited vulnerabilities are unknown to defenders. The challenge is therefore to capture and filter these attack trails from network traffic, connected devices, and target applications, and develop defense mechanisms that can effectively leverage such data to disrupt ongoing attacks and prevent future attempted exploits. Specifically, *anomaly-based* intrusion detection systems alert administrators when deviations from a model of *normal* behavior is detected in the observed data.

[3] *Denning (1987)*

[4] *Sager (2014); DiMaggio (2015); Jeng (2015); Novetta Threat Research Group (2016)*

However, despite its great promise, the advancement of anomaly-based intrusion detection approaches has been hindered by the scarcity of realistic, current, publicly available cyber attack data sets, and the difficulty of accurately and efficiently labeling such data sets, which are often prohibitively large and complex. This *data drought* problem has frustrated comprehensive, timely training of intrusion detection systems (IDSes), thereby raising IDS false alarm generation rates and elevating their susceptibility to attacker evasion techniques.[1]

[1] *Bhuyan et al. (2014); Chandola et al. (2009); Garcia-Teodoro et al. (2009); Patcha and Park (2007); Sommer and Paxson (2010)*

Toward alleviating this data drought, this chapter proposes and examines a novel, decep-tion-based approach to enhancing IDS data streams for faster, more accurate, and more timely evolution of intrusion detection models to emerging attacks and attacker strategies. Deception has long been recognized as a key ingredient of effective cyber warfare,[2] but its applications to IDS have heretofore been recognized exclusively in contexts where the deception is isolated and separate from the data stream in which intrusions must be detected. A typical example is the use of dedicated honeypots to collect attack-only data streams.[3] Such approaches unfortunately have limited training value in that they often mistrain IDSes to recognize only attacks against honeypots, or only attacks by unsophisticated adversaries unable to identify and avoid honeypots. For example, attacks that include substantial interactivity are typically missed, since the honeypot offers no legitimate services.

[2] *cf., Yuill et al. (2006)*

[3] *Vasilomanolakis et al. (2015)*

Our work overcomes this limitation by integrating deceptive attack response capabilities directly into live, production server software. Such honey-patching (see Chapter 2) opens the door to realizing mutually supportive defensive mechanisms that enhance IDS data streams by coordinating capabilities found in multiple layers of the software stack. These *deception-enhanced* data streams quench data drought by providing IDSes with remarkably concept-relevant, current, feature-filled information with which to detect and prevent sophisticated, targeted attacks.

We demonstrate the potential effectiveness of this new IDS approach through the design, implementation, and analysis of DeepDig (DEcEPtion DIGging), a framework for deception-enhanced intrusion detection. Evaluation shows that extra information harvested through potentially successful deceptions (1) improves the precision of anomaly-based IDSes by feeding back into the classifier attack traces, (2) provides feature-rich, multi-dimensional attack data for classification (3) can detect exploits previously unseen by defenders. Our goal in this work is not to evaluate whether deceptions succeed, but rather to assess whether successful deceptions are helpful for intrusion detection, and to what degree. Given the present scarcity of good, current intrusion data sets and the costs of conducting large-scale empirical data collection, we believe that the approach's facility for generating richer, automatically-labeled attack data streams offers exceptional promise for future IDS research.

Our contributions can be summarized as follows:

- We propose a software patching methodology that closes security vulnerabilities in a way that naturally modulates the attack labeling and feature extraction process for intrusion detection. Our approach thus exposes existing, yet unexplored threat information that can be easily gathered through cooperation with the application layer through honey-patching.

- We present a feature-rich attack classification approach that supports better, more accurate characterization of malicious activities, and is resistant against attacker evasion strategies.

- To harness training and test data, we present the design of a framework for synthetic generation of realistic web traffic, which statistically mutates and injects attacks into the generated output streams.

- We implement and evaluate our ideas on large-scale, realistic network- and system-level events generated by this test bed.

4.1 APPROACH OVERVIEW

We first outline practical limitations of traditional machine learning techniques for anomaly detection, motivating our research. We then overview our approach for automatic attack labeling and feature extraction via honey-patching.

4.1.1 *Anomaly-based Intrusion Detection*

Anomaly-based intrusion detection systems flag deviations from expected system behavior, with the foundational assumption that malicious activities exhibit properties that are abnormal relative to legitimate usage of a system.[1] Typically, such systems make use of machine learning techniques (e.g., information theory,[2] neural networks,[3] clustering,[4] or genetic algorithms[5]) to capture a model of normal activity and to find non-conforming patterns in audit data, such as in network packets, system call traces, and application logs.

[1] *Denning (1987)*

[2] *Lee and Xiang (2001)*

[3] *Zhang et al. (2001)*

[4] *Sequeira and Zaki (2002)*

[5] *Sinclair et al. (1999)*

Despite the increasing popularity of machine learning in intrusion detection applications, its success in operational environments has been hampered by specific challenges that arise in the cyber security domain. Fundamentally, machine learning algorithms perform better at identifying similarities than at discovering previously unseen outliers, yet typical intrusion decision scenarios require finding these outliers. A common problem therefore arises from the conventional approach of training a classifier solely with legitimate examples—a setting certainly not ideal for intrusion detection, as it necessitates a perfect model of normality for any reliable classification.[6]

[6] *Sommer and Paxson (2010)*

Feature extraction[7] is also unusually difficult in intrusion detection contexts because security-relevant features are often not known by defenders in advance. The task of selecting appropriate features to detect an intrusion (e.g., features that generate the most distinguishing intrusion patterns) often creates a bottleneck in building effective models, since it demands empirical evaluation. Identification of attack traces among collected workload traces for constructing realistic,

[7] *Blum and Langley (1997)*

unbiased training sets is particularly challenging. Current approaches usually require manual analysis aided by expert knowledge,[1] which severely reduces model evolution and update capabilities to cope with attacker evasion strategies.

[1] *Chandola et al. (2009); Bhuyan et al. (2014)*

A third obstacle is analysis of encrypted data. Encryption is widely employed to prevent unauthorized users from accessing sensitive data transmitted through network links or stored in file systems. However, since network-level anomaly detectors typically discard cyphered data, their efficacy is greatly reduced by the widespread use of encryption technologies.[2] In particular, attackers benefit from encrypting their malicious payloads, making it harder for standard classification strategies to distinguish attacks from normal activity.

[2] *Garcia-Teodoro et al. (2009)*

High false positive rates are another practical challenge for adoption of anomaly detection systems.[3] Raising too many alarms renders anomaly detection meaningless in most cases, as actual attacks are often lost among the many alarms. Studies have shown that effective intrusion detection therefore demands very low false alarm rates.[4]

[3] *Patcha and Park (2007)*

[4] *Axelsson (1999)*

These significant challenges call for the exploration and development of new, accurate anomaly detection schemes that lift together information from many different layers of the software stack. Toward this end, our work extends anomaly-based intrusion detection with the capability to effectively and efficiently detect malicious activities bound to the application layer, affording anomaly detection approaches an inexpensive tool for automatically and continuously extracting security-relevant features for attack detection.

4.1.2 *Digging Deception-Enhanced Threat Data*

DeepDig is a new approach to enhance anomaly-based intrusion detection with threat data sourced from honey-patched applications. Figure 4.1 shows an overview of the approach. Unlike conventional anomaly-based detection approaches, DeepDig incrementally builds a model of *legitimate* and *malicious* behavior based on audit streams and attack traces

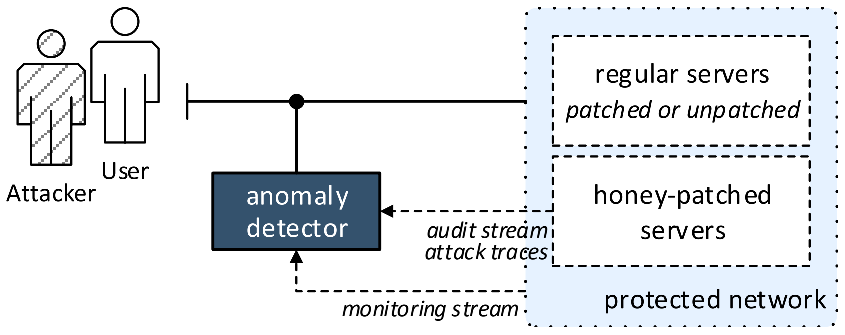


Figure 4.1: DeepDig approach overview

collected from honey-patched servers. This augments the anomaly detector with security-relevant feature extraction capabilities not available to typical network intrusion detectors.

Such capabilities are transparently built into the framework, requiring no additional developer effort (apart from routine patching) to convert the target application into a potent feature extractor for anomaly detection. Since traces extracted from decoys are *true* evidence of malicious activity, this results in an effortless labeling of the data and supports the generation of higher-accuracy detection models.

Honey-patches add a layer of deception to confound exploits of known (patchable) vulnerabilities. Previously unknown (i.e., zero-day) exploits can be mitigated through IDS cooperation with the honey-patches. For example, a honey-patch that collects identifying information about a particular adversary seeking to exploit a known vulnerability can convey that collected information to train an anomaly detector, which can then potentially identify the same adversary seeking to exploit a previously unknown vulnerability.

Our central insight is that software security patches can be repurposed in an IDS setting as automated, application-level feature extractors aggressively maintained by the collective expertise of the software development community. Honey-patching transduces that collective expertise into a highly accurate, rapidly co-evolving feature extraction module for an IDS. The extractor can effortlessly detect previously unseen payloads that exploit known vulnerabilities at the application layer. These can be prohibitively difficult to detect by a strictly network-level IDS due to the challenges summarized in §4.1.1.

By living inside servers that offer legitimate services, our deception-enhanced IDS can target attackers who use one payload for reconnaissance but reserve another for their final attacks. The facility of honey-patches to deceive such attackers into divulging the latter is useful for training the IDS to identify the final attack payload, which can divulge attacker strategies and goals not discernible from the reconnaissance payload alone. The defender's ability to thwart these and future attacks therefore derives from a synergy between the application-level feature extractor and the network-level anomaly detector to derive a more complete model of attacker behavior.

4.2 ARCHITECTURE

DeepDig's architecture, depicted in Figure 4.2, embodies this approach by leveraging application-level threat data gathered from attacker sessions misdirected to decoys. Within this framework, developers use honey-patches to misdirect attackers to decoys that automatically collect and label monitored attack data. The anomaly detector consists of an *attack modeling* component that incrementally updates the anomaly model data generated by honey-patched servers, and an *attack detection* component that uses this model to flag anomalous activities in the monitored perimeter.

MONITORING & THREAT DATA COLLECTION

The decoys into which attacker sessions are forked are managed as a pool of continuously monitored Linux containers.[1] Each container follows the life cycle depicted in Figure 4.3. Upon attack detection, the honey-patching mechanism *acquires* the first available container from the pool. The acquired container holds an attacker session until (1) the session is

[1] *LXC (2014)*

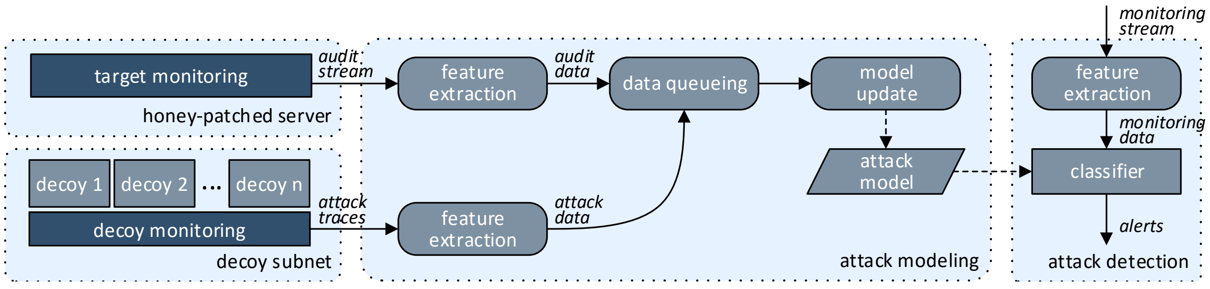


Figure 4.2: DeepDig system architecture overview

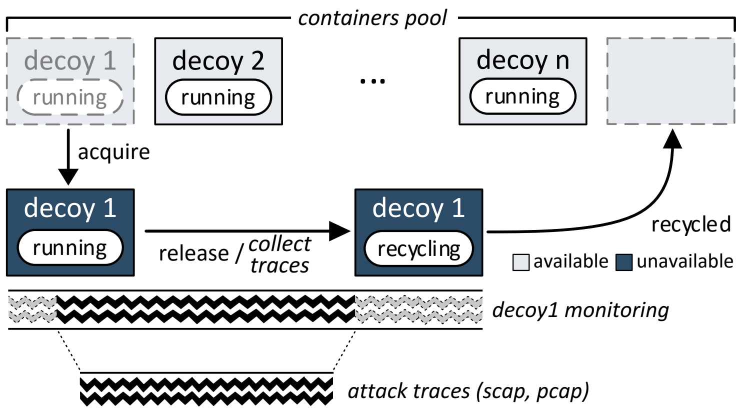


Figure 4.3: Decoy lifecycle and attack traces collection

deliberately closed by the attacker, (2) the connection's *keep-alive* timeout expires, (3) the ephemeral container crashes, or (4) a session timeout is reached. The last two conditions are common outcomes of successful exploits. In any of these cases, the container is released back to the pool and undergoes a recycling process before becoming available again.

After decoy *release*, the *container monitoring component* extracts the session trace (delimited by the acquire and release timestamps), labels it, and stores the trace outside the decoy for subsequent feature extraction. Decoys only host attack sessions, so precisely collecting and labeling their traces (both at the network and OS level) becomes effortless.

DeepDig distinguishes between three separate input data streams: (1) the *audit stream*, collected at the target honey-patched server, (2) *attack traces*, collected at decoys, and (3) the *monitoring stream*, the actual test stream collected from regular servers. Each of these streams contains network packets and operating system events captured at each server environment. To minimize performance impact, we used two powerful and highly efficient software monitors: *sysdig* (to track system calls and modifications made to the file system), and *tcpdump* (to monitor ingress and egress of network packets). Specifically, the decoys into which attacker sessions are forked store monitored data outside the decoy environments to avoid possible tampering with the collected data.

ATTACK MODELING & DETECTION Using the continuous audit stream and incoming attack traces as labeled input data, DeepDig incrementally builds a machine learning model that captures legitimate and malicious behavior. The raw training set (composed of both audit stream and attack traces) is piped into a feature extraction component that selects relevant, non-redundant features (see §4.3) and outputs feature vectors—*audit data* and *attack data*—that are grouped and queued for subsequent model update. Since the initial data streams are labeled and have been preprocessed, feature extraction becomes very efficient and can be performed automatically. This process repeats periodically according to an administrator-specified policy. Finally, the *attack detection* module uses the most recently constructed attack model to detect malicious activity in the run-time *monitoring data*.

4.3 ATTACK DETECTION

To evaluate whether a successful deception is helpful for intrusion detection, we have designed and implemented two feature set models to facilitate anomaly detection: (1) *Bi-Di* detects anomalies in security-relevant network streams, and (2) *N-Gram* finds anomalies in system call traces. Our framework does not impose any particular classification approach.

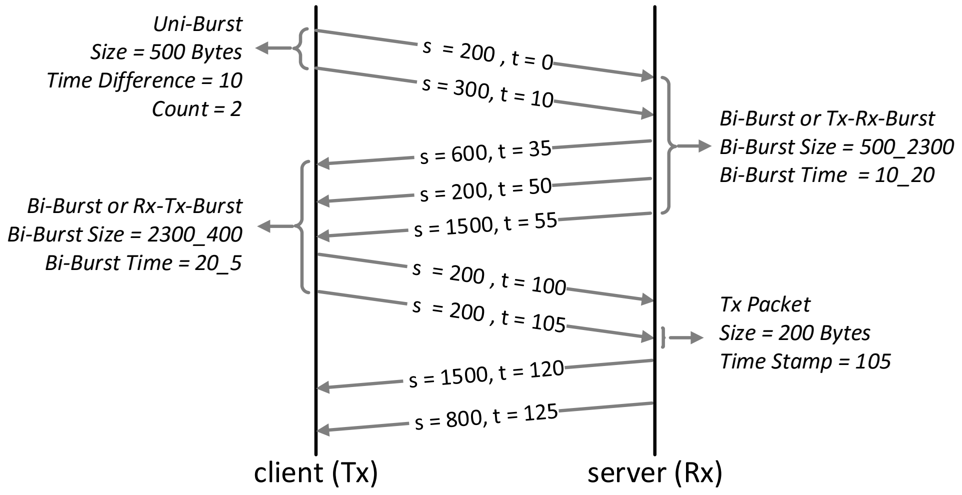


Figure 4.4: Example of uni- and bi-directional bursts

4.3.1 *Network Packet Analysis*

Bi-Di is a packet-level network behavior analysis approach that extracts features from sequences of packets and *bursts*—consecutive packets oriented to the same direction (*viz.*, uplinks from client to server, or downlinks from server to client). It uses distributions from individual burst sequences (*uni-bursts*) and sequences of two adjacent bursts (*bi-bursts*). Extracted features include statistics about burst size and time. To be robust against encrypted payloads, we limit feature extraction to packet headers.

Figure 4.4 illustrates packet flows between client (*Tx*) and server (*Rx*). *Tx packets* and *Rx packets* are uplink and downlink packets, respectively. Bi-Di constructs histograms using features extracted from packet lengths and directions, and uses a support vector machine (SVM)[1] for classification. To overcome dimensionality issues associated with burst sizes, *bucketization* is applied to group bursts into correlation sets (e.g., based on frequency of occurrence). We next detail the features selected for uni- and bi-bursts.

[1] *Cortes and Vapnik (1995)*

Uni-burst features include burst *size*, *time*, and *count*—i.e., the sum of the sizes of all packets in the burst, the amount of

Table 4.1: Packet, uni-burst, and bi-burst features

Category	Features
Packet (Tx/Rx)	Packet length
Uni-Burst (Tx/Rx)	Uni-Burst size Uni-Burst time* Uni-Burst count
Bi-Burst (Tx-Rx/Rx-Tx)	Bi-Burst size* Bi-Burst time*

*new features introduced in this work

time for the entire burst to be transmitted, and the number of packets it contains, respectively. Taking direction into consideration, one histogram for each are generated.

Bi-burst features include time and size attributes of *Tx-Rx-bursts* and *Rx-Tx-bursts*. Each is comprised of a consecutive pair of downlink and uplink bursts. The size and time of each are the sum of the sizes of the constituent bursts, and the sum of the times of the constituent bursts, respectively.

Table 4.1 summarizes the features used in our approach. It highlights new features proposed for uni- and bi-bursts as well as features proposed in prior works.[1]

[1] *Alnaami et al. (2015); Dyer et al. (2012); Hintz (2003); Panchenko et al. (2011); Wang et al. (2014)*

Bi-bursts capture dependencies between consecutive packet flows in a TCP connection. Based on connection characteristics, such as network congestion, the TCP protocol applies flow control mechanisms (e.g., window size and scaling, acknowledgement, sequence numbers) to ensure a level of consistency between Tx and Rx. This influences the size and time of transmitted packets in each direction. Each packet flow (uplink and downlink) thereby affects the next flow or burst until communicating parties finalize the connection.

4.3.2 *System Call Analysis*

The monitored data also includes system streams comprising a collection of OS events, where each event contains multiple fields including event type (e.g., *open*, *read*, *select*), process name, and direction. Our prototype implementation was developed for Linux x86_64 systems, which exhibit about 314 distinct possible system call events. DeepDig builds histograms from these system calls using N-Gram—a system-level approach that extracts features from system call traces. N-Gram uses SVM for classification.

There are four feature types: *Uni-events* are system calls, and can be classified as enter or exit events. *Bi-events* are sequences of two consecutive events, where system calls in each bi-event constitute features. There are therefore $\binom{314}{2}$ possible features in a worst-case scenario. Similarly, *tri-* and *quad-events* are sequences of three and four consecutive events (respectively).

To overcome the inherent dimensionality problem of this approach, N-Gram uses a tree-based feature selection algorithm[1] to select the most effective features out of a pool of possible dimensions and discard the irrelevant ones. The algorithm builds several decision trees, each of which calculates information gain (using the computed entropy) for each feature. The features that have the highest average information gain are selected. The rest are discarded as they introduce noise to the data set.

[1] *Genuer et al. (2010)*

Bi-Di and N-Gram differ in feature granularity; the former uses coarser-grained bursting while the latter uses only individual system call co-occurrences.

4.3.3 *Learning*

CLASSIFICATION Bi-Di and N-Gram both use SVM for classification. Using a convex optimization approach and mapping non-linearly separated data to a higher dimensional linearly separated feature space, SVM separates positive (attack) and negative (benign) training instances by a hyperplane with the maximum gap possible. Prediction labels are assigned based on which side of the hyperplane each monitoring/testing instance belongs.

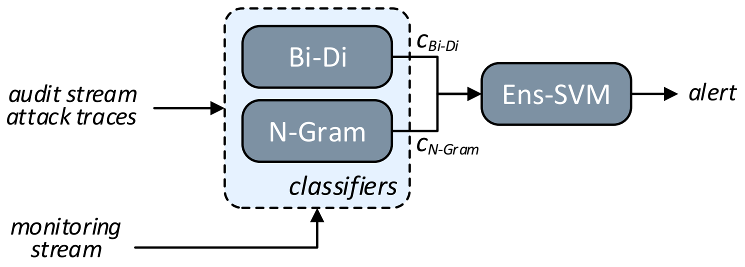


Figure 4.5: Ensemble of Bi-Di and N-Gram

ENS-SVM Bi-Di and N-Gram can be combined to obtain a better predictive model. A naïve approach concatenates features extracted by Bi-Di and N-Gram into a single feature vector and uses it as input to the classification algorithm. However, this approach has the drawback of introducing normalization issues. Alternatively, *ensemble methods* combine multiple classifiers to obtain a better classification outcome via majority voting techniques. For our purposes, we use the ensemble depicted in Figure 4.5. This *Ens-SVM* ensemble classifies new input data by weighting the classification outcomes of Bi-Di and N-Gram based on their individual accuracy indexes.

Figure 4.6 describes the voting approach for Ens-SVM. For each instance in the monitoring stream, if both Bi-Di and N-Gram agree on the predictive label (line 7), Ens-SVM takes the common classification as output (line 8). Otherwise, if the classifications disagree, Ens-SVM takes the prediction with the highest SVM confidence (line 10). Confidence is rated using Platt scaling,[1] which uses the following sigmoid-like function to compute the classification confidence:

[1] *Platt (1999)*

$$P(y = 1|x) = \frac{1}{1 + \exp\left(Af(x) + B\right)} \qquad (4.1)$$

where y is the label, x is the testing vector, $f(x)$ is the SVM output, and A and B are scalar parameters learned using Maximum Likelihood Estimation (MLE). This yields a probability measure of how much a classifier is confident about assigning a label to a testing point.

```
Data: training data: $TrainX$, testing data: $TestX$
Result: a predicted label $\mathcal{L}_{\mathfrak{I}}$ for each testing instance $\mathfrak{I}$
begin
    $\mathbb{B} \leftarrow \text{updateModel}(\text{Bi-Di}, TrainX)$;
    $\mathbb{N} \leftarrow \text{updateModel}(\text{N-Gram}, TrainX)$;
    for each $\mathfrak{I} \in TestX$ do
        $\mathcal{L}_{\mathbb{B}} \leftarrow \text{label}(\mathbb{B}, \mathfrak{I})$;
        $\mathcal{L}_{\mathbb{N}} \leftarrow \text{label}(\mathbb{N}, \mathfrak{I})$;
        if $\mathcal{L}_{\mathbb{B}} == \mathcal{L}_{\mathbb{N}}$ then
            $\mathcal{L}_{\mathfrak{I}} \leftarrow \mathcal{L}_{\mathbb{B}}$;
        else
            $\mathcal{L}_{\mathfrak{I}} \leftarrow \text{label}\left(\underset{c \in \{\mathbb{B}, \mathbb{N}\}}{\arg\max}\, \text{confidence}(c, \mathfrak{I}),\ \mathfrak{I}\right)$;
        end
    end
end
```

Figure 4.6: *Ens-SVM* voting approach

4.4 IMPLEMENTATION

We developed an implementation of DeepDig for 64-bit Linux (kernel 3.11 or above). It consists of two main components: the monitoring controller and the attack detection component.

The monitoring controller provides the server monitoring and attack trace extraction capabilities from decoys. It consists of about 150 lines of JavaScript code, and leverages *tcpdump*, *editcap*, and *sysdig* for network and system call tracing and preprocessing.

The attack detection component is implemented as two Python modules: The feature extraction module, comprising about 1200 lines of code using *scikit-learn*[1] for data preprocessing and tree-based feature selection; and the classifier component, comprising 230 lines of code that references the *Weka*[2] wrapper for LIBSVM.[3] The source-code modifications required to honey-patch vulnerabilities in Apache HTTP, Bash, PHP, and OpenSSL consist of about 35 lines of C code.

[1] *Pedregosa et al. (2011)*

[2] *Hall et al. (2009)*

[3] *Chang and Lin (2011)*

4.5 EVALUATION

This section demonstrates the practical advantages and feasibility of the deception-enhanced intrusion detection capabilities of DeepDig. First, we present our approach for generating realistic web traffic to emulate normal and malicious user behavior, which we harness to automatically generate training and test datasets for our experiments. Then, we discuss our experimental setup and investigate the effects of different attack classes and varying numbers of attack instances on the predictive power and accuracy of anomaly detection. Finally, we assess the performance impact of the deception monitoring scheme that captures network packets and system events.

All experiments were performed on a 16-core host with 24 GB RAM running 64-bit Ubuntu 14.04 (Trusty Tahr). Regular and honey-patched servers were deployed as LXC containers running atop the host using the official LXC Ubuntu template.

4.5.1 *Synthetic Traffic Generation*

Public network packet capture repositories for IDS evaluation are very scarce, and the few datasets publicly available are either outdated (e.g., 1998/2000 DARPA IDS datasets) or they omit packet payloads (e.g., CAIDA[1]). Furthermore, finding datasets that satisfy scenario-specific requirements (e.g., containing particular attack classes) is a well-recognized problem in this domain.[2]

To overcome these limitations, we built a synthetic traffic generator and testing harness to evaluate our approach. Figure 4.7 shows an overview of our traffic generation framework, inspired by prior work.[3] It synthetically streams *encrypted* legitimate and malicious workloads onto a honey-patched server, resulting in labeled audit streams and attack traces (collected at decoys) for training set generation.

[1] *CAIDA (2016)*

[2] *Ahmed et al. (2016); Bhuyan et al. (2014)*

[3] *Boggs et al. (2014)*

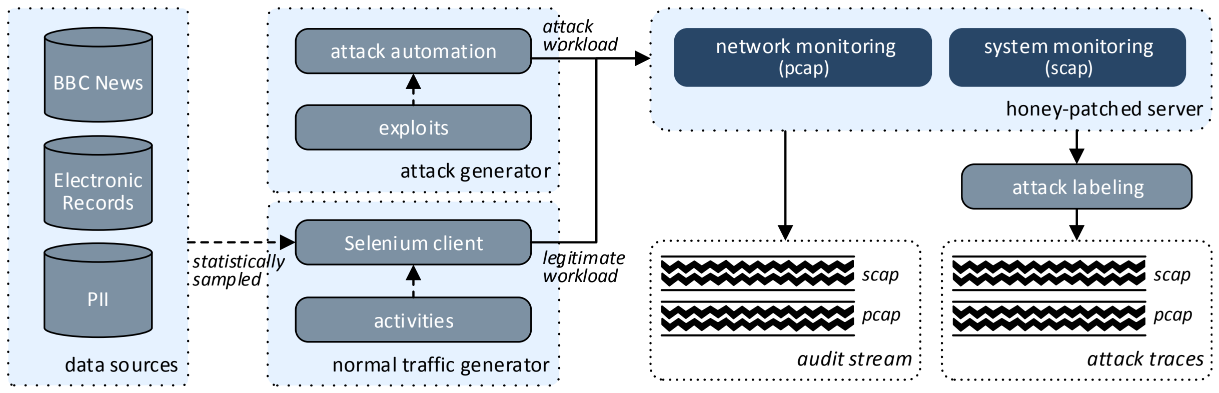


Figure 4.7: Synthetic traffic generation and testing harness

LEGITIMATE DATA GENERATION Normal traffic is created by automating complex user actions on a typical web application, leveraging *Selenium*[1] to automate user interaction with a web browser (e.g., clicking buttons, filling out forms, navigating a web page). We generated traffic for 10 different user activities (each repeated 200 times), including web page browsing, e-commerce website navigation, blog posting, and interacting with a social media web application. The setup included a CGI web application and a PHP-based Wordpress application hosted on a monitored Apache web server. To enrich the set of user activities, the Wordpress application was extended with *Buddypress* and *Woocommerce* plugins for social media and e-commerce web activities, respectively.

[1] *Selenium (2016)*

To create realistic interactions with the web applications, our framework feeds from online data sources, such as the BBC text corpus, online text generators for personally identifiable information (e.g., usernames, passwords), and product names to populate web forms. To ensure diversity, we statistically sampled the data sources to obtain user input values and dynamically generated web content. For example, blog title and body is statistically sampled from the BBC text corpus, while product names are picked from the product names data source.

ATTACK DATA GENERATION Attack traffic is generated based on real vulnerabilities. For this evaluation, we selected 12 exploits for four well-advertised, high-severity vulnerabilities. These include CVE-2014-0160 (Heartbleed[1]), CVE-2014-6271 (Shellshock[2]), CVE-2012-1823 (improper handling of query strings by PHP in CGI mode), and CVE-2011-3368 (improper URL validation). In addition, nine attack variants exploiting CVE-2014-6271 were created to carry out different malicious activities, such as leaking password files and invoking bash shells on the remote web server. These vulnerabilities are important as attack vectors because they range from sensitive data exfiltration to complete control and remote code execution. To emulate realistic attack traffic, we interleaved attacks and normal traffic following the strategy of Wind Tunnel.[3]

[1] *Codenomicon (2014)*

[2] *NIST (2014b)*

[3] *Boggs et al. (2014)*

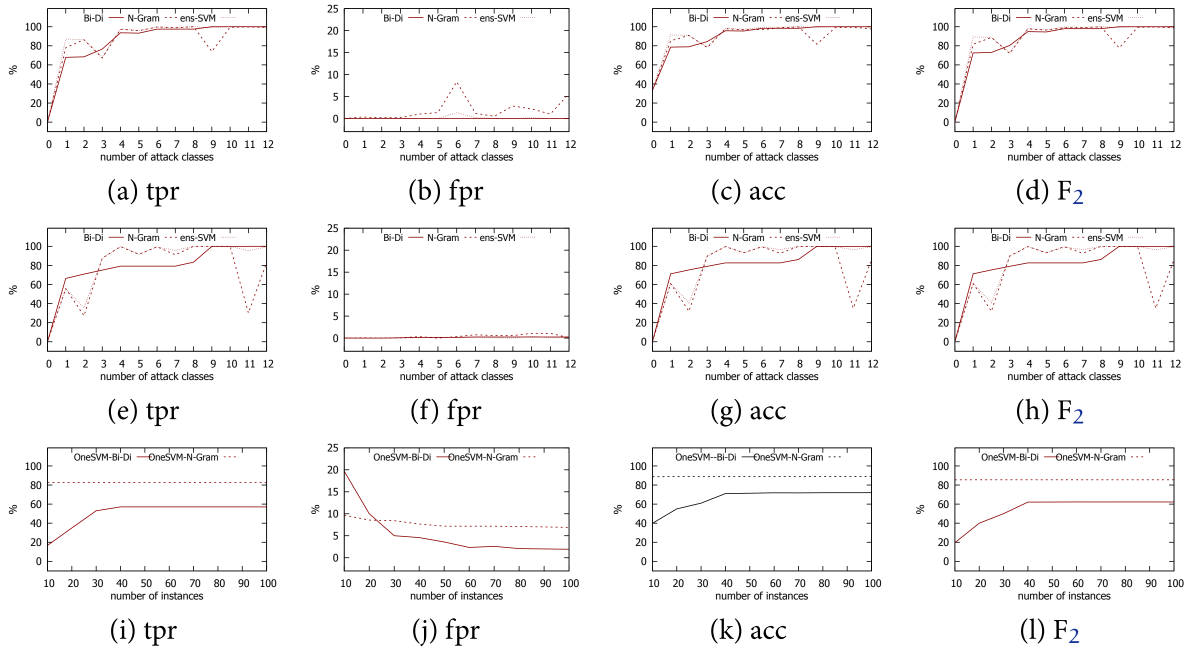
DATASET Table 4.2 summarizes the synthetic data generated for our experimental evaluation. The synthetic traffic generator was deployed on a separate host to avoid interference with the test bed server. In total, we generated 12 GB of (uncompressed) network packets and system events over a period of three weeks. After feature extraction, the audit data comprised 1200 normal instances and 1200 attack instances. Monitoring data consisted of 2400 normal and attack instances gathered at unpatched web servers, where the distribution of normal and attack instances varies per experiment.

4.5.2 *Experimental Results*

Using this dataset, we trained the classifiers presented in §4.3 and assessed their individual performance against test streams containing both normal and attack workloads. In each experiment, we measured the true positive rate (*tpr*), false positive rate (*fpr*), accuracy (*acc*), and F_2 score of the classifier, where the F_2 score is interpreted as the weighted average of the precision and recall, reaching its best value at 1 and worst at 0.

Table 4.2: Summary of synthetic data generation.

Normal workload summary		
Activity	**Application**	**Description**
Post	CGI web app	Posting blog on a guestbook CGI web app
Post	Wordpress	Posting blog on wordpress
Post	Wordpress buddypress plugin	Posting comment on social media web app
Registration	Wordpress woocommerce plugin	Product registration and product description
Ecommerce	Wordpress woocommerce plugin	Ordering of a product and checkout
Browse	Wordpress	Browsing through a blog post
Browse	Wordpress buddypress	Browsing through a social media page
Browse	Wordpress woocommerce plugin	Browsing product catalog
Registration	Wordpress	User registration
Registration	Wordpress woocommerce plugin	Coupon registration
Attack workload summary		
Attack Type	**Attack action**	**Description**
CVE-2014-0160	Information leak	Openssl vulnerability
CVE-2012-1823	System remote hijack	PHP CGI vulnerability
CVE-2011-3368	Port scanning	Apache vulnerabilty
CVE-2014-6271	System hijack (7 variants)	Bash vulnerability
CVE-2014-6271	Remote Password file read	Bash vulnerability
CVE-2014-6271	Remote root directory read	Bash vulnerability



(a) tpr (b) fpr (c) acc (d) F_2

(e) tpr (f) fpr (g) acc (h) F_2

(i) tpr (j) fpr (k) acc (l) F_2

Figure 4.8: Accuracies of Bi-Di, N-Gram, and Ens-SVM for increasing numbers of attack classes in the training set for (a)–(d) validation on decoy data, and (e)–(h) validation on unpatched server data. (i)–(l) Baseline evaluation for OneSVM-Bi-Di and OneSVM-N-Gram.

detection accuracy To evaluate the accuracy of intrusion detection, we tested each classifier after incrementally training it with increasing numbers of attack classes. Each class consists of 100 distinct variants of a single exploit, as described in §4.5.1, and an n-class model is one trained with 1–n attack classes. For example, a 3-class model is trained with 300 attacks from 3 different classes. In each run, the classifier is trained with 1200 normal instances and $n \in [1, 12]$ attack classes.

validation on decoy data The first experiment measures the accuracy of each classifier against a test set composed of 1200 normal instances and 1200 uniformly distributed attack instances gathered at decoys. Figure 4.8(a)–(d) presents the results, which serve as a preliminary check that the classifiers can accurately detect attack instances resembling the ones comprised in their initial training set.

TESTING ON UNPATCHED SERVER DATA The second experiment also measures each classifier's accuracy, but this time the test set was derived from monitoring streams collected at regular, *unpatched* servers, and having a uniform distribution of attacks. Figure 4.8(a)–(d) shows the results, which indicate that the anomaly detection models of each classifier generalize beyond data collected in decoys. This is critical because it demonstrates the classifier's ability to detect previously unseen attack variants. DeepDig thus enables administrators to add an additional level of protection to their entire network, including hosts that cannot be promptly patched, via the adoption of a honey-patching methodology.

The results also show that as the number of training attack classes increases—which are proportional to the number of vulnerabilities honey-patched—a steep improvement in the true positive rate of both classifiers is observed, reaching an average *tpr* of above 99% for the compounded Ens-SVM, while average false positive rate in all experiments remained below 0.03%. This demonstrates the positive impact of the *feature-enhancing* capabilities of deceptive application-level attack responses like honey-patching.

BASELINE EVALUATION This experiment compares the accuracy of our anomaly detection approach to the accuracy of an unsupervised outlier detection strategy, which is commonly employed in typical intrusion detection scenarios, where labeling attack data is not feasible or prohibitively expensive. For this purpose, we implemented two *One-class SVM* classifiers, *OneSVM-Bi-Di* and *OneSVM-N-Gram*, using Bi-Di and N-Gram models for feature extraction, respectively. We do not use Ens-SVM here, since One-class SVM does not provide confidence or probability estimates with its predictions.

One-class SVM uses an unsupervised approach, where the classifier trains on one class and predicts whether a test instance belongs to that class, thereby detecting *outliers*—test instances outside the class. To perform this experiment, we incrementally trained each classifier with an increasing number of *normal* instances, and tested the classifiers after each iteration against the same test set used in the previous experiment. The results presented in Figure 4.8(i)–(l) clearly highlight critical limitations of conventional outlier intrusion detection systems: reduced predictive power, lower tolerance to noise in the training set, and higher false positive rates.

In contrast, our supervised approach overcomes such disadvantages by automatically streaming onto the classifiers labeled security-relevant features, without any human intervention. This is possible because honey-patches identify security-relevant events at the point where such events are created, and not as a separate, *post-mortem* manual analysis of traces.

FALSE ALARMS To evaluate the fpr-reducing effects of DeepDig, we trained each classifier with data sets containing 1200 normal instances and 0–30 attack instances per class of attack in 30 incremental training iterations. We tested each classifier after every iteration step and plotted the results in Figure 4.9. Observe that with just a few attack instances (≈ 5 per attack class), the false positive rates of all classifiers dropped to close to zero percent, demonstrating DeepDig's continuous feeding back of attack samples onto classifiers greatly reduces false alarms.

4.5.3 *Base Detection Analysis*

In this section we measure the success of DeepDig in detecting intrusions in the realistic scenario where attacks are a small fraction of the interactions. Although risk-level attribution for cyber attacks is difficult to quantify in general, we use the results of a recent study[1] to approximate the probability of attack occurrence for the specific scenario of targeted attacks against business and commercial organizations. The

[1] *Dudorov et al. (2013)*

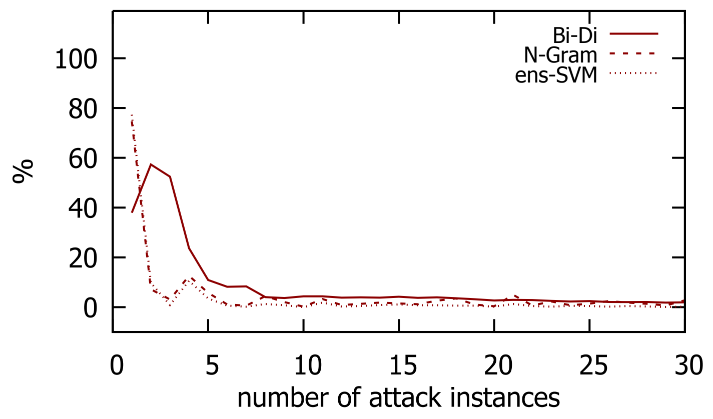


Figure 4.9: False positive rate for various training sets

study's model assumes a determined attacker leveraging one or more exploits of known vulnerabilities to penetrate a typical organization's internal network, and approximates the *prior* of a directed attack to $P_A = 6\%$ (using threat statistics from 2011).

To estimate the success of intrusion detection, we use a *base detection rate* (*bdr*), expressed using the Bayes theorem as:

$$P(A|D) = \frac{P(A)\ P(D|A)}{P(A)\ P(D|A) + P(\neg A)\ P(D|\neg A)]} \tag{4.2}$$

where A and D are random variables denoting the occurrence of a targeted attack and the detection of an attack by the classifier, respectively. We use *tpr* and *fpr* as approximations of $P(D|A)$ and $P(D|\neg A)$, respectively. To obtain *tpr* and *fpr* values for this scenario, we repeated our first experiment, changing the distribution of legitimate and attack instances in the monitoring stream to match the study's model.

Table 4.3 presents the accuracy values and *bdr* for each classifier, assuming $P(A) = P_A$. The numbers expose a practical problem in intrusion detection research: Despite having high accuracy values, typical intrusion detection systems are rendered ineffective when confronted with their staggering low base detection rates. This is in part due to their intrinsic

Table 4.3: Base detection rate percentages for an approximate targeted attack scenario ($P_A \approx 6\%$)[1]

[1] *Dudorov et al. (2013)*

Classifier	*tpr*	*fpr*	*acc*	*F*$_2$	*bdr*
OneSVM-Bi-Di	58.3	6.37	71.2	63.1	36.9
OneSVM-N-Gram	93.1	8.23	92.6	93.5	41.9
Bi-Di	87.9	0.26	98.5	89.4	95.5
N-Gram	95.1	0.44	99.1	95.1	93.3
Ens-SVM	98.3	0.19	99.7	98.3	96.9

inability to eliminate false positives in operational contexts. In contrast, the *fpr*-reducing properties of our framework—i.e., the ability to suppress false alarms through automatic labeling of network- and system-level attack features—affords the construction of anomaly detection systems that can detect intrusions much more effectively in realistic settings.

4.5.4 *Monitoring Performance*

To assess the performance overhead of DeepDig's monitoring capabilities, we used *ab* (Apache HTTP server benchmarking tool) to create a massive user workload (more than 5,000 requests in 10 threads) against two web server containers, one deployed with network and system call monitoring and another unmonitored.

Figure 4.10 shows the results, where web server response times are ordered in an ascendingly. Our measurements show average overheads of $0.2\times$, $0.4\times$, and $0.7\times$ for the first 100, 250, and 500 requests, respectively, which is expected given the heavy workload profile imposed on the server. Since server computation accounts for only about 10% of overall web site response delay in practice,[2] this corresponds to observable overheads of about 2%, 4%, and 7% (respectively).

[2] *Souders (2007)*

While such overhead characterizes feasibility, it is irrelevant to deception because unpatched, patched, and honey-patched servers are all slowed equally by the monitoring activity.

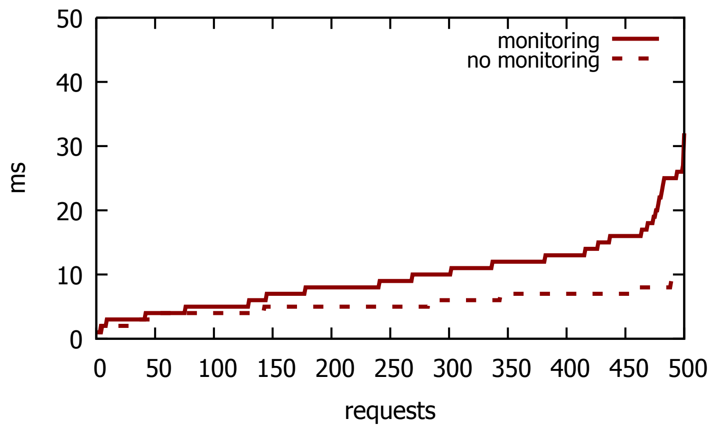


Figure 4.10: DeepDig performance overhead

The overhead therefore does not reveal which apparent vulnerabilities in a given server instance are genuine patching lapses and which are deceptions, and it does not distinguish honey-patched servers from servers that are slowed by any number of other factors (e.g., fewer computational resources).

4.6 DISCUSSION

METHODOLOGY Our experiments show that just a few strategically chosen honey-patched vulnerabilities accompanied by a equally small number of honey-patched applications provide a machine learning-based IDS sufficient data to perform substantially more accurate intrusion detection, thereby enhancing the security the entire network. Thus, we arrive at one of the first demonstrable measures of value for deception in the context of cyber security: its utility for enhancing IDS data streams.

MORE FEATURES One avenue of future work is to leverage system call arguments in addition to the features we collected. A common technique is to use pairwise similarity between arguments (as sequences) of different streams,[1] and then implement a k-NN (k-Nearest Neighbors) algorithm with

[1] *Chandola et al. (2009)*

longest common subsequence (LCS) as its distance metric. Generally, packet- and system-level data are very diverse and contain many more discriminating features that could be explored.

ONLINE TRAINING The flood of data that is continuously streamed into a typical IDS demands methods that support fast, online classification. Prior approaches update the classification model incrementally using training batches consisting of one or more training instances. However, this strategy necessitates frequently re-training the classifier, and requires a significant number of instances per training. Future research should investigate the appropriate conditions for re-training the model. *Change point detection* (CPD)[1] is one promising approach to determine the optimal re-training predicate, based on a dynamic sliding window that tracks significant changes in the incoming data, and therefore resists concept-drift failures.

[1] *Haque et al. (2016)*

SUPERVISED LEARNING Our approach facilitates supervised learning, whose widespread use in the domain of intrusion detection has been impeded by many challenges involving the manual labeling of attacks and the extraction of security-relevant features.[2] Our results demonstrate that the language-based, active response capabilities provided via application-level honey-patches significantly ameliorates both of these challenges. The facility of deception for improving other machine-learning based security systems should therefore be investigated.

[2] *Chandola et al. (2009)*

GENERALIZATION The results presented in §4.5 show that our approach substantially improves the accuracy of anomaly detection, reducing false alarms to much more practical levels. Although we used many variations of well-known attacks and showed how DeepDig generalizes when increasing the pool of attacks, future work should explore larger numbers of attack classes to simulate threats to high-profile targets. Due to to the high-dimentional nature of the collected data, we chose SVM in Bi-Di and N-Gram. Linearly separating such data is complicated by various feature interactions, such as network burst sequences and system IO events. SVM is suitable for this task as it maps non-linear data points to another linearly separable feature space using the *kernel trick*.

INTRUSION DETECTION DATASETS One of the major challenges in evaluating intrusion detection systems is the dearth of publicly available datasets, which is often aggravated by privacy and intellectual property considerations. To mitigate this problem, security researchers often resort to synthetic dataset generation, which affords the opportunity to design test sets that validate a wide range of requirements. Nonetheless, a well-recognized challenge in custom dataset generation is how to capture the multitude of variations and features manifested in real-world scenarios.[1]

[1] *Bhuyan et al. (2014)*

Our evaluation approach builds on recent breakthroughs in synthetic dataset generation for IDS evalaution[2] to create statistically representative workloads that resemble realistic web traffic, thereby affording the ability to perform a synthetic, yet meaningful evaluation of IDS frameworks.

[2] *Boggs et al. (2014)*

EVALUATION Establishing a straight comparison of our results to prior work can be very challenging. The majority of anomaly-based intrusion detection techniques are still tested on extremely old datasets.[3] For instance, recently-proposed SVM-based approaches for network intrusion detection have reported true positive rates in the order of 92% for the DARPA/KDD datasets, with false positive rates averaging 8.2%.[4] Using the model discussed in §4.5.3, this corresponds to an approximate base detection rate of only 42%, in contrast to 96.9% estimated for our approach. However, such comparison can lead to erroneous conclusions, as the assumptions made by DARPA/KDD do not reflect the contemporary attack protocols and recent vulnerabilities targeted by our model.

[3] *Ahmed et al. (2016); Sommer and Paxson (2010)*

[4] *Manandhar and Aung (2014); Zhang et al. (2015)*

4.7 CONCLUSION

This chapter introduced, implemented, and evaluated a new approach for enhancing anomaly-based IDSes with threat data sourced from deceptive, application-layer, software traps. Unlike conventional anomaly-based detection approaches, DeepDig incrementally builds models of legitimate and malicious behavior based on audit streams and traces collected from these traps. This augments the IDS with inexpensive and automatic security-relevant feature extraction capabilities. These capabilities require no additional developer effort apart from routine patching activities. This results in an effortless labeling of the data and supports a new generation of higher-accuracy detection models.

5

DECEPTION AS A SERVICE

Cloud computing has attracted significant attention in recent years as a model for scalable service consumption and as a delivery platform for service-oriented computing. Revolutionary advances in hardware and virtualization technologies have elevated cloud computing to a thriving industry that affords enterprises the ability to shrink IT expenditures, adapt quickly to variable workloads, and reduce administration overhead. These successes have been achieved principally by reinventing a wide variety of traditionally on-site computing resources as deliverable services according to the mantra *Everything as a Service* (XaaS).[1] Pillars of the XaaS mantra typically include Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

[1]*Banerjee et al. (2011)*

The central observation of this chapter is that the technological advances at the heart of the cloud computing movement have now converged to cultivate a remarkably fertile ground for mass-scale cyber deception as a defense. In particular, many foundational cloud technologies including massive replication, high performance process migration and load balancing, hardware and software heterogeneity, aggressive multitenancy, and multi-layer virtualization, have led to computing environments ideal for assembling a "hall of mirrors" in which legitimate services are interlaced with deceptive computations, platforms, data, and software, all designed to misdirect attackers away from valuable targets. We refer to this vision as *Deception as a Service* (DaaS).

DaaS comes as a welcome contrast to widespread fears about the security of cloud computing systems. With over a third of our personal data projected to be stored in public clouds by 2016,[1] it is unsurprising that the great majority of cloud stakeholders have reported fears of cloud-specific attacks by external threat actors.[2] To help alleviate these concerns, there has been increasing research on enhancing the security of cloud platforms. Advancements in this area include ensuring computation integrity,[3] protecting data security and privacy,[4] and intrusion detection and prevention.[5]

[1] *Gartner (2012)*

[2] *Burger et al. (2013)*

[3] *Khan and Hamlen (2012b); Santos et al. (2009); Chow et al. (2009)*

[4] *Takabi et al. (2010); Bowers et al. (2009); Nepal et al. (2011); Khan and Hamlen (2012a); Pearson (2009)*

[5] *Khan and Hamlen (2013); Khan et al. (2014)*

DaaS complements and enhances these advancements by introducing new, deception-powered defenses that leverage facilities and opportunities unique to cloud environments, and that cannot be realized as effectively on traditional computing platforms. Clouds are thus championed as a cyber security *opportunity*, rather than a security-resistant environment to which traditional defenses must be transitioned.

Throughout the remainder of the chapter, we present a preliminary investigation of the power of commodity cloud computing systems for enhancing and strengthening the deceptive capabilities of honey-patching (see Chapter 2). Our approach enables the automatic deployment and scaling of RedHerring on multiple cloud architectures and environments, affording cyber-defenders a new form of active, offensive response to attacks in commodity cloud and service-oriented infrastructures.

5.1 ARCHITECTURAL OVERVIEW

Modern computing environments typically require the configuration and orchestration of multiple services for applications to function. These can range from a few instances (e.g., a web server and a database), to very complex setups such as IaaS deployments requiring many components to be installed, configured, and interconnected (e.g., OpenStack). To ease the task of creating and maintaining such service-oriented environments, configuration management tools like Chef[6] and Puppet,[7] or even general-purpose scripting languages such as

[6] *Chef (2016)*

[7] *Puppet (2016)*

Python or Bash, automate the configuration of machines to a particular specification.

More recently, Juju[1] has been introduced as a model specification for service oriented architectures and deployments, enabling transparent and efficient management of cloud services on both public cloud infrastructures (e.g., Amazon EC2, Microsoft Azure, Joyent Triton) and private infrastructures (e.g., OpenStack, physical servers, containers). Juju abstracts and simplifies cloud deployment and scaling, and provides users with client-side command-line tools to uniformly manage locally and remotely deployed services. Application-specific knowledge such as dependencies, operational events like backups and upgrades, and integration options with other pieces of software are encapsulated in Juju's *charms*. A charm defines everything required to deploy a particular service, and is composed of user-implemented *hooks* which Juju invokes at different stages of the service's lifecycle.

[1] *Canonical (2016)*

A SERVICE-ORIENTED ARCHITECTURE FOR HONEY-PATCHING Using Juju as underlying framework, we implemented a charm that automates the deployment and scaling of RedHerring on top of IaaS environments, therefore augmenting cloud infrastructures with DaaS capabilities through honey-patching. Figure 5.1 shows an overview of the DaaS architecture. Users of our platform can easily deploy RedHerring on a variety of environments including KVM, Xen, and LXC. Physical deployment is also supported through bare-metal containers and metal-as-a-service, which lets physical servers be treated like virtual machines in the cloud. For example, the command line instruction

```
juju-deploy redherring -to machine:0/lxc
```

instructs the *juju state service* component to deploy a new unit of RedHerring on *environment 2*. This triggers the automatic instantiation of a new container (i.e., *machine 0*), and the *juju agent* running on the container is tasked with the execution of the charm specification.

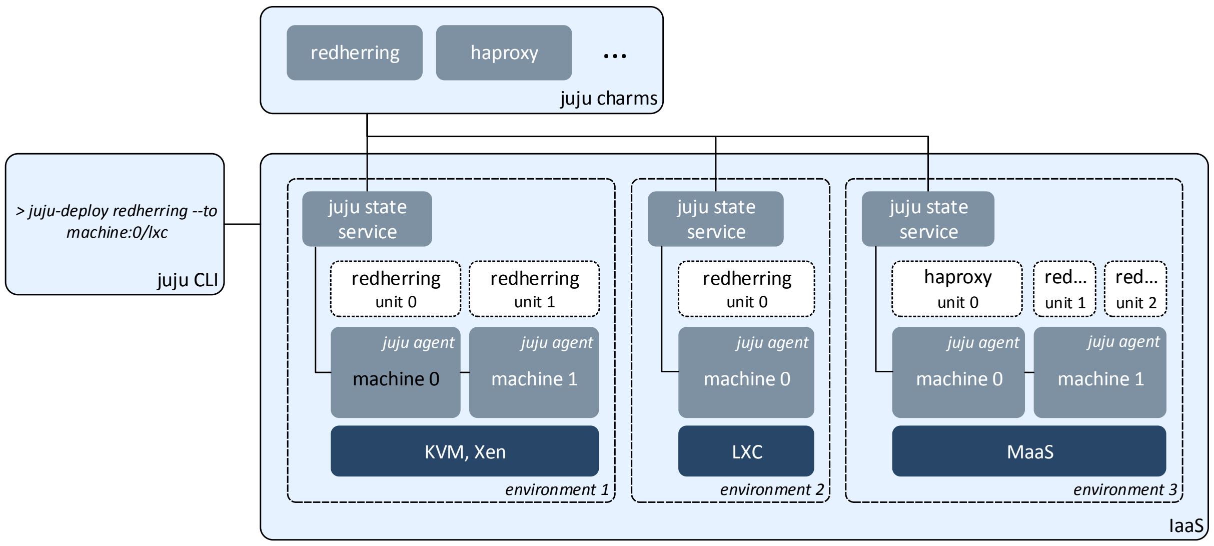


Figure 5.1: Overview of a DaaS architecture leveraging honey-patching

SCALABILITY One of the main benefits of this service-oriented architecture is the simplicity of scaling services up and down. For example, to scale RedHerring up horizontally, users first instruct juju to add the desired number of units to the existent deployment (e.g., `juju add-unit redherring -to machine:1/maas`), and then setup load balancing to distribute the work load among units. To achieve this, one option is to use the infrastructure's built-in load balancing capabilities. An alternative option is to deploy a load balancing service such as *HAProxy*:

```
juju-deploy haproxy -to machine:0/maas;
juju add-relation redherring haproxy
```

Conversely, scaling down follows a similar procedure to remove deployed units.

5.2 SERVICE MODELING

Each RedHerring service instance goes through a series of events during its lifecycle: *install*, *configure*, *start*, *upgrade*,

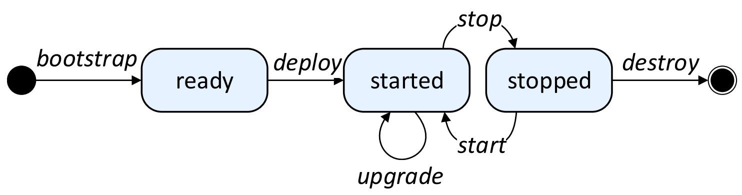


Figure 5.2: RedHerring service modeling for a DaaS deployment

and *stop*. Figure 5.2 depicts these events and the associated state transitions. Two special events, *bootstrap* and *destroy*, result in pre-defined actions executed by Juju, and correspond to the creation and destruction of the deployment environment, respectively. For each of the remaining events, Juju executes specific hooks specified in the redherring charm. Hooks are executable scripts in a charm's hooks directory, and are invoked by the unit's juju agent at particular times in the service lifecycle. We designed redherring hooks to be *idempotent*, meaning that there is no observable difference between running a hook once and running it multiple times in a row.

Figure 5.3 details hooks associated with RedHerring's deployment. Hooks *setup target environment* and *setup decoy environment* pre-installs onto the deployment environment the target and decoy containers file systems according to the charm's configuration parameters (e.g., string prefix for decoy names, container pool size). Hook *install applications* fetches all applications specified in the configuration (e.g., a honey-patched Apache HTTP) and installs them into the target container. Finally, hooks *install reverse proxy* and *setup internal network* install the proxy on the unit and isolate the target container from the pool of decoys as a separate subnet, respectively.

5.3 EXPERIMENTAL VALIDATION

To validate our DaaS architecture, we have implemented a Juju charm for RedHerring, and used it to deploy honey-patching

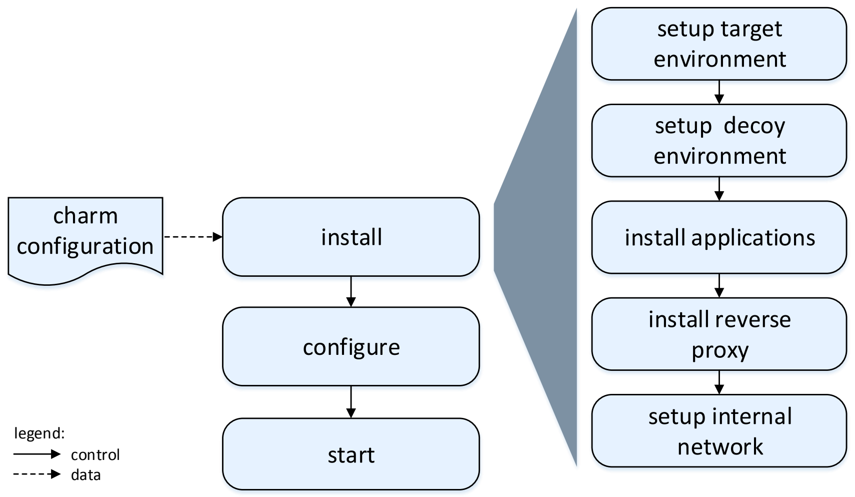


Figure 5.3: Redherring charm deployment hooks

as a service both locally (on LXC containers running on top of a Linux VM) and remotely (on public clouds, including Amazon EC2 and Joyent Triton). To test our deployments, we streamed into each instance synthetic attacks derived from our regression test suite. Overall, our hooks consist of about 460 lines of Bash code, and expose a rich set of configuration parameters to ease deployment customization. Assessing how the unique set of features and characteristics of each cloud provider affects deception delivery in public clouds is planned for future work. In particular, we plan to investigate architectural properties that will facilitate the implementation of multi-layer deception strategies across the deception stack.

5.4 CONCLUSION

This chapter introduced and implemented Deception as a Service (DaaS) as new security model for cloud computing. DaaS transparently enhances existing cloud infrastructures with deceptive capabilities through honey-patching, offering cyber-defenders a new form of active, offensive response to attacks in commodity cloud and service-oriented infrastructures.

6

MOVING TARGET DECEPTION THROUGH VERSION EMULATION

Effective deception often demands high dynamicity—defenders must remain adaptable and agile in order to effectively lure and misdirect diverse, evolving adversaries.[1] However, the honey-patching, process image secret redaction, and DaaS techniques presented in Chapters 2, 3, and 5 (respectively) have heretofore been presented in relatively static configurations. For example, the decision of which vulnerabilities to honey-patch, which secrets to redact, and which deceptive services to deploy on the cloud, have all been presented according to semi-manual processes dependent upon human intervention. While helpful for presentational clarity, this static approach is suboptimal for waging long-term deceptive campaigns, since it could result in deceptions becoming predictable and stale, potentially affording attackers the opportunity and time to fingerprint and circumvent honey-patched applications. In addition, as time elapses and new vulnerabilities emerge, attacks and probing activity change,[2] potentially rendering old deceptions less enticing to cyber criminals.

The research presented in this chapter was performed in part while the author was visiting IBM T.J. Watson Research Center during the Summer of 2015, in collaboration with K.W. Hamlen, M.P. Stoecklin, J. Jang, and X. Hu. Author Araujo conducted all of the research reported herein.

[1] *Heckman et al. (2015, Chapter 6)*

[2] *Arbaugh et al. (2000); Lippmann et al. (2002); Rescorla (2005); Browne et al. (2001)*

To overcome this disadvantage, this chapter proposes *software deception steering* as a new moving target defense technique for counterreconnaissance and attack intelligence gathering, which leverages application-level, deceptive attack responses through honey-patching to constantly adapt the *deception surface* of the target application. Toward this end, we designed and implemented QUICKSAND, an adaptive, software version emulation architecture, in which the set of honey-patched vulnerabilities in a target application is dynamically re-selected to increase the likelihood of deceiving and entrapping attackers. Based on the history of past attacks (e.g., gathered at the network level), QUICKSAND chooses to emulate a particular software version henceforth, with a particular set of vulnerabilities honey-patched (and all other known vulnerabilities regular-patched). This *moving* deception surface undermines the attacker's ability to identify and detect specific configurations of honey-patched applications (e.g., by probing for a particular set of honey-patches), therefore rendering honey-patching more resilient against fingerprinting attacks.

Our work includes the following contributions:

- We propose a deception-based moving target architecture to dynamically honey-patch software applications, rendering them less predictable and more robust against attackers' anti-deception efforts.

- Our work leverages existing enterprise defense infrastructures for threat intelligence gathering, and proposes a new class of cyber maneuvers for moving target defense based on honey-patching; our approach benefits from previously gathered attack data to drive honey-patch re-selection in order to increase the likelihood of deceiving and misdirecting attackers.

- We propose, design, and implement an effective version-control strategy to facilitate patch re-selection and automatically resolve source-level conflicts between patches.

6.1 SYSTEM OVERVIEW

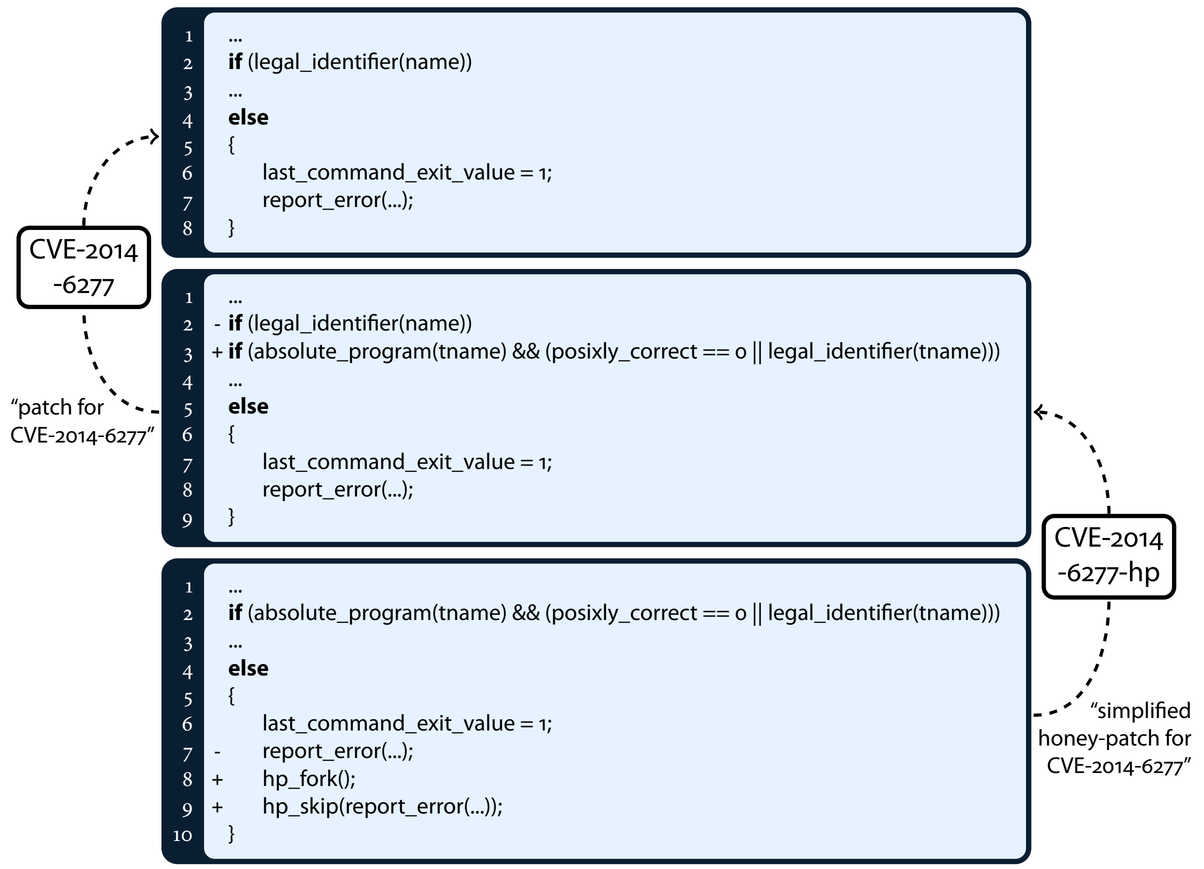
6.1.1 *Software Version Emulation*

We define *software version emulation* as a cyber-deception maneuver for honey-patching. Leveraging intrusion alerts collected at the network perimeter, QUICKSAND dynamically adapts the target application to emulate a particular software version, with a particular set of vulnerabilities honey-patched (and all other known vulnerabilities regular-patched), a particular set of features/modules enabled, and a particular guest OS version deployed in decoy environments. The scope of adaptation can go beyond the application and host boundaries—for instance, perimeter defenses (if any) can also be reconfigured to intentionally allow previously filtered attacks to reach the honey-patch. This reconfiguration need not happen live; it can be re-selected during nightly reboots, for example. The selections are based on which configuration is likely to gather the most useful threat data given the history of past attacks.

6.1.2 *Design Principles*

Software version emulation requires a patch management framework that facilitates software version composition and minimizes the occurrence of source-level conflicts between patches. QUICKSAND defines honey-patches as modifications to their corresponding regular, vendor-supplied patches. For instance, Figure 6.1 exemplifies a vulnerability causing the GNU Bash shell to improperly parse function definitions in the values of environment variables.[1] Prior to the patch, the vulnerable shell interpreter allowed remote attackers to execute arbitrary code or cause a denial of service on the victim's machine. The patch, named CVE-2014-6277 in this example, fixes the vulnerability by extending the check for what constitutes a legal function identifier to include some extra sanity checks (line 2–3 in the patch code, depicted in diff style). The honey-patch CVE-2014-6277-hp modifies the original patch to fork attacks onto decoy environments while impersonating the unpatched code (lines 8–9 in the honey-patch code) in order to deceive adversaries. Encoding honey-patches in this manner naturally models the dependency among honey-patches, their corresponding patches

[1] *NIST (2014a)*



```
...
if (legal_identifier(name))
...
else
{
    last_command_exit_value = 1;
    report_error(...);
}
```

```
 ...
- if (legal_identifier(name))
+ if (absolute_program(tname) && (posixly_correct == 0 || legal_identifier(tname)))
 ...
 else
 {
     last_command_exit_value = 1;
     report_error(...);
 }
```

```
  ...
  if (absolute_program(tname) && (posixly_correct == 0 || legal_identifier(tname)))
  ...
  else
  {
      last_command_exit_value = 1;
-     report_error(...);
+     hp_fork();
+     hp_skip(report_error(...));
  }
```

CVE-2014 -6277
"patch for CVE-2014-6277"
CVE-2014 -6277-hp
"simplified honey-patch for CVE-2014-6277"

Figure 6.1: Patch and honey-patch for CVE-2014-6277 (abbreviated), and dependencies between them denoted by dashed arrows

and unpatched source code. It also makes patch/honey-patch pairs conflict-free by construction, greatly simplifying the task of composing new versions of the target application.

Patch dependencies (denoted by dashed arrows between two patches) are calculated based on how patches affect source code rather than by the order in which they are introduced into the code base. This removes the temporal constraint among patches and enables the selection of patch sets based on their *true* dependencies. This patch dependency model is implemented in the Darcs version control system,[1] which our software version-emulation architecture leverages to select consistent, conflict-free application versions for deployment.

[1] *Roundy (2005)*

6.1.3 *Darcs Patch Theory*

Darcs is a *change-based* version control system. In contrast to conventional history-based version control systems (e.g., Subversion, Git, CVS), which represent repository states as file trees, the state of a Darcs repository is defined by the set of patches it contains. This facilitates a specific kind of *cherry-picking* operation—one that is not constrained by temporal dependencies among patches—which is central to our patch set selection model. Before explaining how cherry-picking works, we introduce a few definitions and properties of Darcs' underlying patch theory.[1]

[1]*Lynagh (2006); Darcs (2016)*

DEFINITIONS The state of a repository is also called a *context*. We write ${}^{o}A^{a}$ to denote that a repository moves from context o to context a via a *patch* A. Patches are normally stored sequentially, and for any consecutive pair of patches, the final state of the first patch must be identical to the initial state of the second patch. A sequence of patches is written in left to right order, such as ${}^{o}A^{a}B^{b}C^{c}$ (or simply ABC if we omit patch contexts). Parallel patches share a common initial context and diverge to two different states ($A \vee B$). Conversely, anti-parallel patches have different initial states yielding the same context ($A^{-1} \wedge B^{-1}$).

INVERSION Every Darcs patch is invertible, affording the application of patches in either forwards or backwards directions to reach a particular context: $(AB)^{-1} = B^{-1}A^{-1}$. In particular, AA^{-1} has no effect, and $(A^{-1})^{-1} = A$.

COMMUTATION The commutation of patches A and B is represented by $AB \leftrightarrow B'A'$, where A' and B' are intended to perform the same change as A and B. Intermediate states may differ however: ${}^{o}A^{a}B^{b} \leftrightarrow {}^{o}B'^{x}A'^{b}$. A *merge* operation is defined as a pairwise commutation, taking two parallel patches and converts them into a pair of sequential patches: $A \vee B \Longrightarrow AB' \leftrightarrow BA'$.

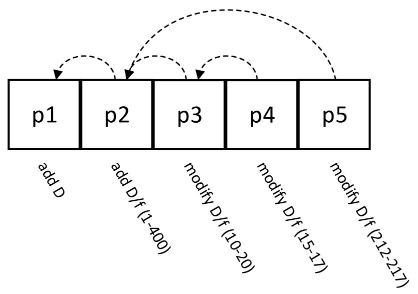


Figure 6.2: A repository state showing patch dependencies

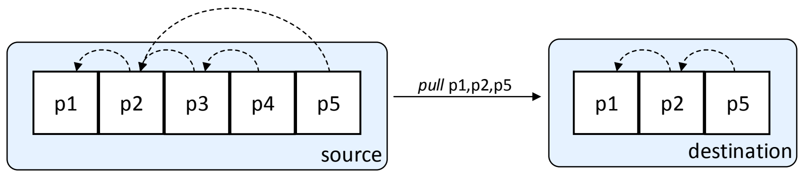


Figure 6.3: Patch cherry picking

CHERRY PICKING Patch *cherry picking* refers to the ability to pull patches from a repository regardless of the order in which they were originally pushed into the repository. To illustrate, consider the repository state depicted in Figure 6.2. The repository consists of patches p1–p5, and the changes made by each patch are summarized underneath each patch. The dependencies between patches (denoted by dashed arrows) are computed by Darcs. Figure 6.3 illustrates cherry picking for this particular example. The result of pulling patches p1, p2, and p5 from the *source* onto the *destination* repository is that the selected patches are adjusted to fit the new context (without p3 and p4). Darcs automatically performs such adjustments using its powerful patch manipulation algebra to provide users of the system the ability to reason about patches as sets—despite patches being stored as sequences internally.

PATCH OBLITERATION AND CONSISTENCY Another advantage of patch commutativity is that patches can be undone (or *obliterated*) without rolling back patches that historically succeed them. In the example above, patch p4

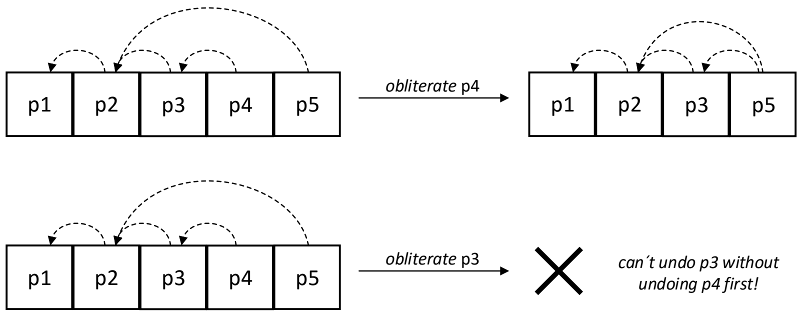


Figure 6.4: Patch obliteration and consistency

can be removed from the repository without undoing p5, as illustrated in Figure 6.4. To accomplish this, Darcs rearranges the sequence of patches by commuting p4 with p5, and then removes p4. However, Darcs does not allow p3 to be removed without first undoing p4; allowing this operation would constitute a patch dependency violation and render the state of the repository inconsistent.

6.2 ARCHITECTURE

The architecture of QUICKSAND is shown in Figure 6.5. The *analysis and correlation* component parses intrusion alerts and correlates them with intrusion signature metadata. The *patch selection* module takes this aggregated data and selects which version of the software should be deployed. The *version deployment* module then uses this information to synthesize and deploy a new version of the application into REDHERRING, including the specification of the target modules and environment. This process executes repeatedly, and the delay threshold can be fixed, random, or dynamically adjusted (e.g., based on evidence and severity of intrusion alerts collected at the network perimeter).

6.2.1 *Patch Management*

Figure 6.6 illustrates our patch management strategy. Regular patches are stored (or *pushed*) into Darcs repositories base

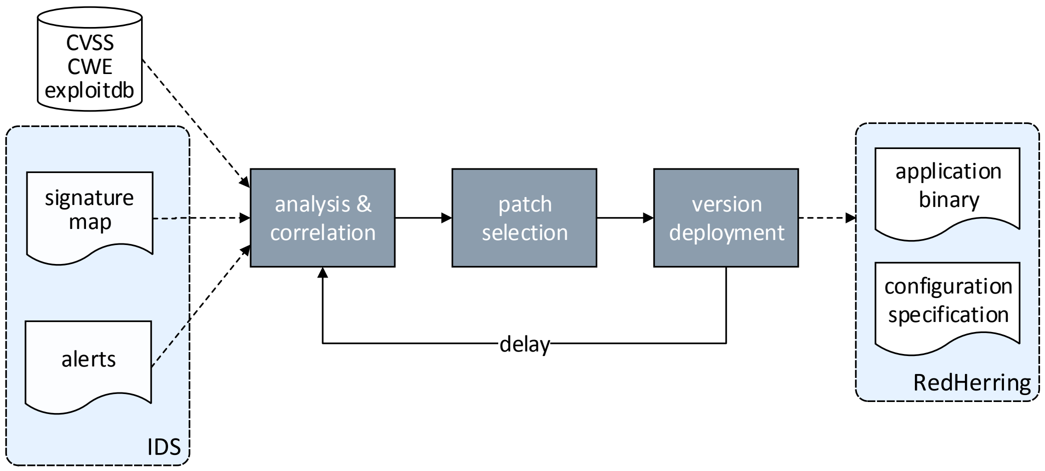


Figure 6.5: QUICKSAND architecture overview

and hp, and honey-patches are stored into repository hp only. We call B the set of patches making up the base version of the software (e.g., the initial commit, a specific tagged version of the application containing all patches up to the tag). Candidate versions selected by the path selection module are stored as tags (e.g., v1–vn) by *pulling* specific patch sets from hp, which allows them to be easily retrieved for version deployment.

This patch management strategy leverages the underlying Darcs infrastructure, which automatically computes the transitive dependency relations for any given patch selection. For example, when pulling honey-patch p4-hp, Darcs correctly pulls patch set B and patches p3, p4, and p4-hp. This has the advantage of enabling a much simpler patch set generation algorithm (see Section 6.2.3).

6.2.2 *Alert Analysis and Correlation*

The three-step alert analysis and correlation workflow is shown in Figure 6.7. First, intrusion alerts are parsed, and each alert class is annotated with descriptive statistics and target information. In the second step, the correlation module parses the intrusion detection system's signature map to

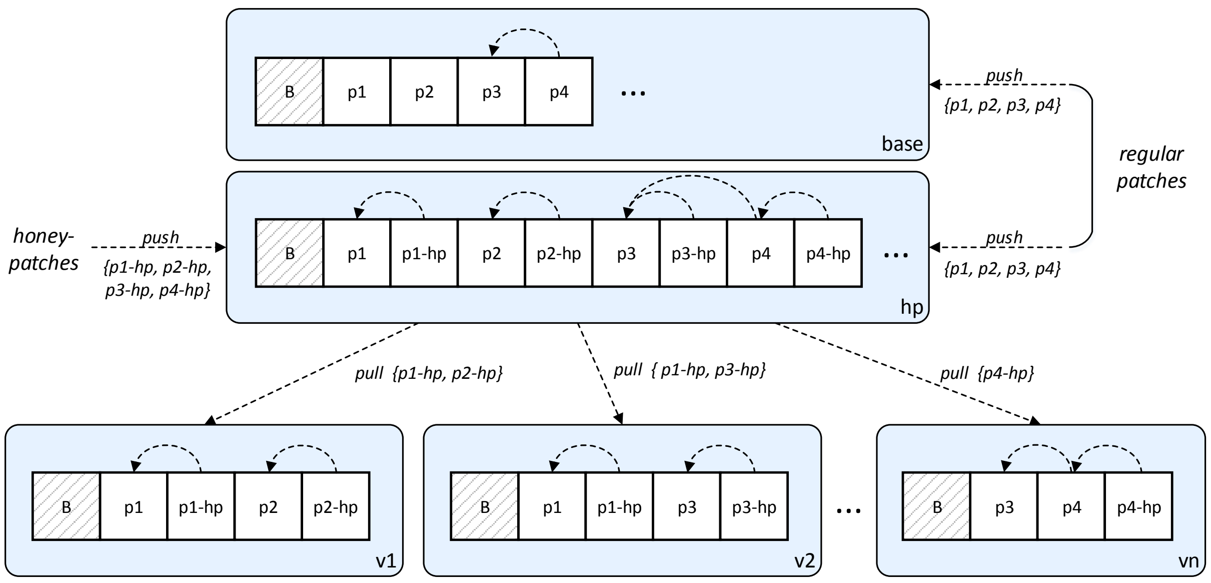


Figure 6.6: QUICKSAND patch management. Patch set B denotes the set of patches making up the base source code of the software; patch dependencies pointing to it have been omitted.

extract the signature information for each alert object and cross-references it with the corresponding CVE identification derived from reference field specified in the alert metadata. This step additionally filters intrusion alerts whose signatures target vulnerabilities that haven't been identified as CVEs. The last step consults vFeed[1] to look up common vulnerability and exploit databases (e.g., CVSS, CWE, exploit-db) in order to aggregate threat intelligence metadata (e.g., vulnerability risk scores, exploit availability) to alert objects, which are used by subsequent modules of the version selection process. Listing 6.1 shows an alert object containing threat metadata for CVE-2014-6277.

[1] *Toolswatch (2016)*

6.2.3 *Patch Selection*

QUICKSAND's patch selection process is shown in Figure 6.8. It comprises two phases: (1) conflict resolution and (2) patch set selection, which are described below.

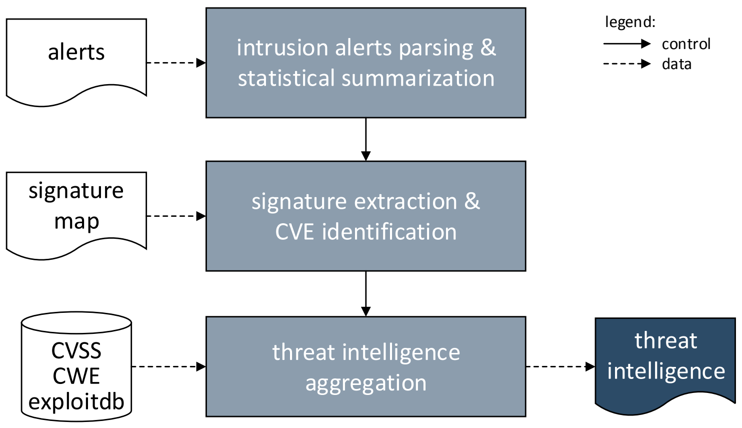

alerts
intrusion alerts parsing & statistical summarization
legend:
control
data
signature map
signature extraction & CVE identification
CVSS
CWE
exploitdb
threat intelligence aggregation
threat intelligence

Figure 6.7: Alert correlation and threat information generation

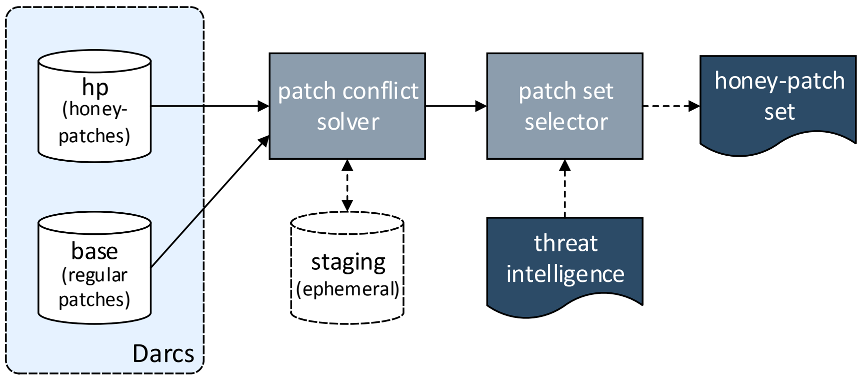


Figure 6.8: Patch conflict resolution and selection

CONFLICT RESOLUTION A conflict in our system is defined by the following syntactic rule: If (honey-)patch $\mathcal{A}$ and (honey-)patch B prescribe different contents for the same line of code, then $\mathcal{A}$ and B cannot coexist automatically in the same version.

Figure 6.9 details (in pseudocode) the algorithm for generating conflict-free patch sets. Its inputs include the set Π of available security patches, the base repository B containing only regular patches, and the repository HP of honey-patches. Lines 1–2 initialize conflict set cs to empty, and initialize a temporary repository Δ as a copy of B. The algorithm then populates cs with all conflicting patch pairs in Π by checking

Listing 6.1: Alert object containing threat metadata for CVE-2014-6277

```
{ cveID: CVE—2014—6277, targets: {('192.168.134.150', 80), ('192.168.134.139', 8080), ...}
  cvssScore: {
            'impact': '10.0',
            'access_complexity': 'low',
            'confidentiality_impact': 'complete',
            'availability_impact': 'complete',
            'authentication': 'none',
            'access_vector': 'network',
            'exploit': '10.0',
            'base': '10.0',
            'integrity_impact': 'complete' },
  cwe: { id: 'cwe—78', term: 'OS␣Command␣Injection' },
  cpe: { 'cpe:/a:gnu:bash:2.02.1', 'cpe:/a:gnu:bash:2.01.1', ...},
  published: 2014—09—24T14:48:04.477—04:00,
  public_exploit: 'yes',
  count: 115 }
```

the result of merging the corresponding honey-patch pair from *HP* into Δ, resetting Δ between each merge operation (lines 3–10). Line 11 removes the temporary repository. Finally, in line 12 the set of conflict-free patch sets is generated by filtering from powerset $\wp(\Pi)$ all patch sets containing any two conflicting patch pairs.

More complex conflict resolution rules can also be specified, at the cost of reducing the space of candidate patch sets for version selection. For example, a more restrictive rule could constrain honey-patches to be applicable only to releases officially reported in the Common Platform Enumerations (CPE) database.

PATCH SET SELECTION Given the set of feasible candidate patch sets generated in the previous step, the patch selection step ranks each patch set according to a user-defined utility function computed over features extracted from metadata

```
Data: $\Pi$: patch set, $B$: base repository, $HP$: honey-patch repository
Result: set of conflict-free patch sets
$cs \leftarrow \emptyset$
$\Delta \leftarrow B$
for $(p_1, p_2) \in \Pi^2$ do
begin
    if $\neg$pull$(\{p_1, p_2\}, HP, \Delta)$ then
    begin
        $cs \leftarrow cs \cup \{(p_1, p_2)\}$
    end
    obliterate('[.*]-hp$', $\Delta$)
end
remove($\Delta$)
return $\{S \in \wp(\Pi) \mid S^2 \cap cs = \emptyset\}$
```

Figure 6.9: Conflict-free patch set generation algorithm

created during alert correlation. These features are grouped into patch features and version features.

Patch features that might influence the effectiveness of candidate patch sets include: (P1) time of patch release, (P2) severity level of patched vulnerability, extracted from CVSS scores, (P3) class of honey-patched vulnerability (e.g., remote code execution, denial of service, confidentiality breach), derived from CWE, CERT, and OWASP categories, (P4) quantity of previously observed, in-the-wild exploits against each honey-patched vulnerability (constrained to some predefined time window of historical data), and (P5) whether there is evidence of publicly available exploits for the particular vulnerability in exploit databases.

Version features lift metrics on individual patches to metrics on sets of patches for version evaluation, and include: (V1) median time of patch release over the candidate patch set, (V2) whether patch-pairs in the candidate set coexisted in nature (i.e., both patches were "current" at the same time), (V3) the percentage of patch-pairs in the candidate set that coexisted, and (V4) the cardinality of the honey-patch subset.

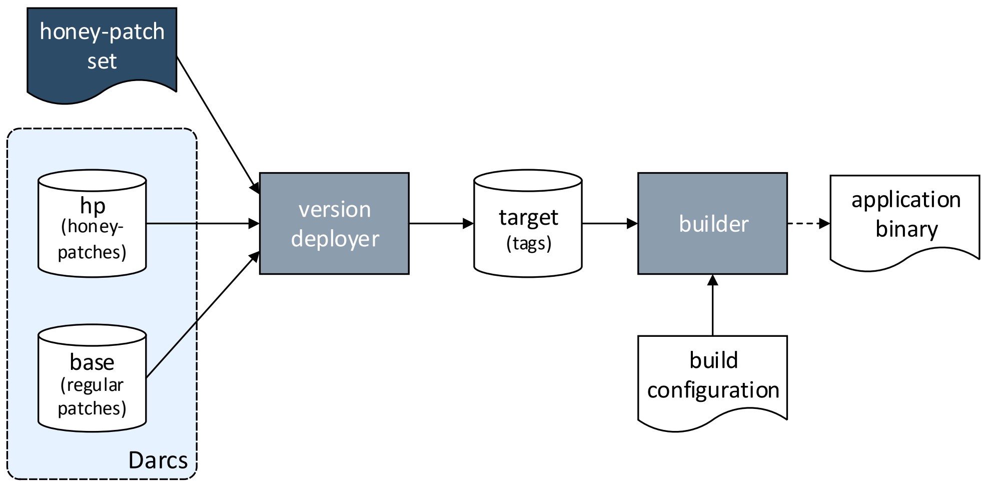


Figure 6.10: Version deployment workflow

6.2.4 *Version Deployment*

Upon completion of patch selection, QUICKSAND deploys a new version of the application into the target environment. Figure 6.10 outlines the steps taken to deploy an application. The first step consists of creating a *working* repository for the application, by first pulling all patches from base into target, and then pulling only the selected honey-patch subset into target. This yields a working repository state that is *tagged* with the selected application version. The final step consists of building the target application from sources, using user-supplied configuration as supplemental input. The configuration parameters are specified per application, as shown in the configuration file illustrated in Listing 6.2, and provide information on how to setup the build environment and release the new application version.

6.3 IMPLEMENTATION

We have developed an implementation of QUICKSAND for the 64-bit version of Linux. The implementation consists of four Python components: the *repository handler* module

Listing 6.2: QUICKSAND configuration file

```
[Apache−1]
app = apache
base_repo = ../data/base
hp_repo = ../data/hp
deploy_repo = ../data/deploy
configure_command = make
install_command = make install
patches = CVE−2014−0160:CVE−2014−6271:CVE−2014−6277:CVE−2014−7169: ...

[Apache−2]
app = apache
...

[OpenSSL]
app = openssl
...
```

consists of about 150 lines of code and wraps Darcs 2.12.0 CLI[1] to offer an API to access the version control system. The *analyzer* component consists of 90 lines of code, and leverages *py-idstools*[2] to parse IDS signature maps and events sourced in unified2 format (a serialized binary stream format specification for IDS events), and *vFeed*[3] to fetch and aggregate threat metadata to alert objects. The *patch selector* module consists of an additional 140 lines of code, and the *version deployment* module adds about 80 lines of code to the system. Our implementation depends on a deployment environment that has been pre-configured with REDHERRING (see Chapter 2).

[1] *Darcs (2016)*

[2] *Py-idstools (2016)*

[3] *Toolswatch (2016)*

6.4 FUTURE WORK

QUICKSAND is an ongoing project, and future work is planned to fully evaluate our software version emulation strategy beyond system prototyping and tests.

EXPERIMENTAL VALIDATION We plan to empirically evaluate our approach through the development of a testing harness that streams the system synthetically generated, labeled attack data derived from real network traffic logs. The labeled data will provide a ground truth to assign performance scores for each software-version generated by QUICKSAND, in a realistic and repeatable manner. Toward this goal, we plan to collect a pool of honey-patched vulnerabilities for highly-targeted server applications and libraries (e.g., Apache, Bash, OpenSSL, OpenSSH, BIND, glibc). For each vendor-supplied security patch, we will craft scripts based on PoC exploits to inject attacks into the regular traffic for the evaluation. Once we have gathered labeled data from our tests, we will extract features from the data set and tune QUICKSAND's version ranking function.

GAME THEORETICAL ANALYSIS To better derive version-adaptation policies that benefit the defender, we plan to model different attack-defense scenarios as adversarial games. The goal is to reduce them to optimization problems that seek to maximize the waste of attackers' resources (e.g., time, difficulty of detecting deception) and minimize the frequency of version deployment (which incurs higher IT costs for defenders). To make the analysis realistic, we envision lifting metrics and mathematical models for attacker probing behavior from prior large-scale empirical studies.[1]

[1] *e.g., Arbaugh et al. (2000); Rescorla (2005); Browne et al. (2001)*

6.5 CONCLUSION

This chapter proposed a new moving target defense technique for counterreconnaissance and attack intelligence gathering, which leverages application-level, deceptive attack responses through honey-patching. Toward this end, we designed and implemented QUICKSAND, an adaptive, software version-emulation architecture, in which the set of honey-patched vulnerabilities in a target application is dynamically re-selected to increase the likelihood of deceiving and entrapping attackers. Based on the history of past attacks, QUICKSAND chooses to impersonate a particular software version using a combination of conflict-free patches and honey-patches, and an endpoint configuration corresponding to the specified version.

7 EXPERIENCES IN ACTIVE CYBER SECURITY EDUCATION

Modern cyber security educational programs that emphasize technical skills often omit or struggle to effectively teach the increasingly important science of cyber deception. A strategy for effectively communicating deceptive technical skills by leveraging honey-patching (see Chapter 2) is discussed and evaluated. Honey-patches mislead attackers into believing that failed attacks against software systems were successful. This facilitates a new form of penetration testing and capture-the-flag style exercise in which students must uncover and outwit the deception in order to successfully bypass the defense. Experiences creating and running the first educational lab to employ this new technique are discussed, and educational outcomes are examined.

This chapter contains material previously published as: F. Araujo, M. Shapouri, S. Pandey, and K.W. Hamlen. "Experiences with honey-patching in active cyber security education." In Proc. of the 8th Workshop on Cyber Security Experimentation and Test, Aug. 2015. Lead author Araujo conducted the majority of the research, leading the lab design, setup, and software creation, and authoring the paper.

7.1 INTRODUCTION

Industrial and governmental demand for employees with superlative, comprehensive cyber security expertise has risen meteorically over the past several years. Cyber security job postings have increased 74% from 2007 to 2013—over double the rate of increase of other IT jobs—and take 24% longer to fill on average than other IT job postings.[1] One reason demand for high expertise is eclipsing supply is the increasing sophistication of threats faced by cyber professionals. In 2014, cyber attacks against large companies rose 40%, yet malicious

[1] *Burning Glass Technologies (2014)*

campaigns are becoming smaller and more efficient, using 14% less email to successfully infiltrate victim networks.[1]

[1] *Lingenheld (2015)*

This underscores a need for effective yet broadly deployable educational strategies for all aspects of cyber security training. Some cyber skills, however, are exceptionally difficult to convey effectively in a classroom setting. A prime example is cyber deception, which is becoming an increasingly central ingredient of many offensive and defensive scenarios.[2] Deceptive social engineering attacks in which attackers impersonate government officials account for over $23,200 in losses *per day* in 2014, according to the FBI Internet Crime Complaint Center.[3] Advanced malware attacks often undertake elaborate user deceptions, such as Stuxnet's replaying of pre-recorded, normal equipment readings to operators at the Natanz nuclear facility during its attack.[4] In light of such practices, the U.S. Air Force has announced cyber deception as a specific focus area for 2015–2016.[5]

[2] *cf., Almeshekah and Spafford (2014); Twitchell (2006); Luo et al. (2011)*

[3] *Internet Crime Complaint Center (IC3) (2015)*

[4] *Langner (2011)*

[5] *U.S. Air Force Materiel Command (2014)*

To raise defender vigilance against deceptive threats, a different way of thinking is required—one that adopts the thinking process of the adversary.[6] Modern defenders must understand the psychology of attackers, and be aware of their strategies and techniques in order to anticipate their actions. In active defense contexts, they require skills for both creating and mitigating deceptive software. Awareness of such issues facilitates development of safer programs, and limits the attack surface exposed to cyber criminals.

[6] *Vigna (2003); Mink and Freiling (2006); Patriciu and Furtuna (2009)*

However, effectively teaching such awareness in a traditional classroom setting can be challenging. Typically, the scholastic experience is contrived, with lectures and assignments following a structured sequence of topics through which students expect to be guided by instructors, and where reading materials provide the theoretical backbone of a rehearsed, time-honored mode of thinking. From a security standpoint, it can be seen as antithetical to most real-world cyber security threat encounters involving advanced adversaries: Modern, targeted cyber threats are often surreptitious, diverse, and unpredictable. Advanced threat-actors are aware of standard educational practices, and therefore adopt strategies that run counter to them. To defend against such threats, future cyber security professionals must be empowered with techniques that can delay reconnaissance efforts, degrade exploitation methods, and confound attackers into moving and acting in a more observable manner.

Among the most promising approaches towards alleviating this problem are Capture the Flag exercises (CTFs), which are commonly organized as competitions where teams score points by exploiting opponents and defending from attacks in real time. However, although such exercises are of great educational value in that they offer lessons not easily taught in a classroom and provide a realistic, safe environment for practicing offensive techniques, they often lack emphasis on *active* cyber defense topics.[1] We believe that ways must be sought to ethically teach students deception and anti-deception techniques in order to make networks more resilient against the emerging wave of advanced threats.

[1] *cf., Heckman et al. (2013)*

Toward this end, we examined honey-patching[2] as a new tool for effectively teaching active defense and attacker-deception to students in the Computer Science Department at The University of Texas at Dallas (UTD). In April of 2015 we organized a small-scale computer lab at UTD to raise awareness about this technique and the broader concepts surrounding cyber deception and anti-deception.[3] The lab was organized with the help of UTD's Computer Security Group (CSG) student association, constituting the first effort to teach honey-patching techniques and strategies outside a research setting. Although small in its size (seven students completed the lab session), the response we obtained from the participants of this early test were extremely positive. Our goal is to relate our experiences to other educators, and to recommend methods and software tools that we have found pedagogically effective for teaching students these important skills. We also plan to leverage this initial experience to contribute larger-scale CTF exercises to major competitions in the future, such as TexSAW (Texas Security Awareness Week), which is held annually at UTD every October.

[2] *Araujo et al. (2014); Araujo and Hamlen (2015)*

[3] *Araujo et al. (2015)*

Listing 7.1: Abbreviated patch for CVE-2014-6271

```
+ if ((flags & SEVAL_FUNCDEF) && command→type != cm_function_def)
+ {
+     internal_warning ("%s:␣ignoring␣function␣definition␣attempt", ...);
+     should_jump_to_top_level = 0;
+     last_result = last_command_exit_value = EX_BADUSAGE;
+     break;
+ }
```

Listing 7.2: Honey-patch for CVE-2014-6271

```
  if ((flags & SEVAL_FUNCDEF) && command→type != cm_function_def)
  {
+     hp_fork();
+     hp_skip(
          internal_warning ("%s:␣ignoring␣function␣definition␣attempt", ...);
          should_jump_to_top_level = 0;
          last_result = last_command_exit_value = EX_BADUSAGE;
          break;
+     );
  }
```

The research reported herein is covered by UTD IRB approval MR15-185. The educational lab and subsequent data analysis were conducted by personnel who are NIH-certified in protection of human research subjects. The lab was organized and overseen by student officers trained in risk management, including ethical and nondiscriminatory treatment of individuals.

7.2 HONEY-PATCHING SHELLSHOCK

In September 2014 we honey-patched the Shellshock GNU Bash remote command execution vulnerability (CVE-2014-6271)[1] within hours of its public disclosure as part of our AFOSR/NSF active defense and attack-attribution research

[1] *NIST (2014b)*

program. Shellshock was one of the most severe vulnerabilities in recent history, affecting millions of then-deployed web servers and other Internet-connected devices. This high impact combined with its ease of exploitation makes it a prime candidate for penetration testing exercises.

Listing 7.1 shows an abbreviated, vendor-released patch in diff style for Shellshock. The patch introduces a conditional that validates environment variables passed to Bash, declining function definition attempts. Prior to this patch, attackers could take advantage of HTTP headers as well as other mechanisms to enable unauthorized access to the underlying system shell of remote targets. This patch exemplifies a common vulnerability mitigation: dangerous inputs or program states are detected via a boolean test, with positive detection eliciting a corrective action. The corrective action is typically readily distinguishable by attackers—in this case, a warning message is generated and the function definition is ignored.

Listing 7.2 presents an alternative, honey-patched implementation of the same patch. In response to a malicious input, the honey-patched application forks itself onto a confined, ephemeral, decoy environment, and behaves henceforth as an unpatched, vulnerable version of the software. Specifically, line 3 forks the user session to a decoy container, and macro `hp_skip` in line 4 elides the rejection in the decoy container so that the attack appears to have succeeded. Meanwhile, the attacker session in the original container is safely terminated (having been forked to the decoy), and legitimate, concurrent connections continue unaffected.

As a result, adversaries attempting to exploit Shellshock in a victim server that has been honey-patched receive server responses that seem to indicate that the exploit has succeeded. However, the shell commands they inject are actually executing in a decoy environment stocked with disinformation for attackers to explore. This provides an ideal environment for students to penetrate as part of exercises focused on cyber deception. Some of their attacks may genuinely hijack the victim server (e.g., those that exploit unpatched vulnerabilities), others observably fail (e.g., those that exploit patched vulnera-

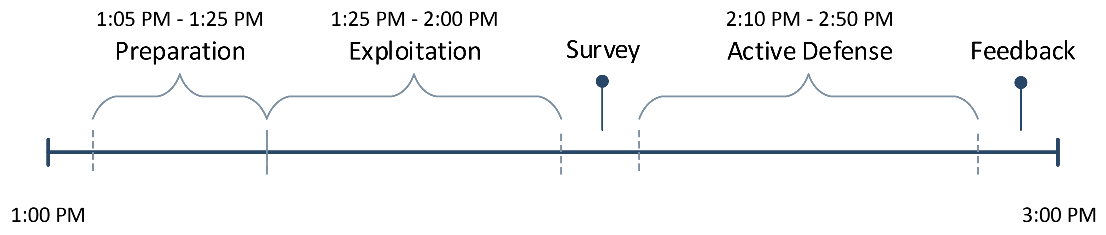


Figure 7.1: Lab timeline and overview.

bilities), while yet others only appear to succeed (e.g., those that exploit honey-patched vulnerabilities). The challenge is to discover that a deceptive outcome exists and counter it.

7.3 LAB OVERVIEW

We organized an open lab with the main goal of educating students on offensive security and active cyber defense concepts using honey-patching as the underlying framework. To boost student expectation and interest, we selected the Shellshock[1] vulnerability as our unit of study. This choice was motivated by the scale and impact of Shellshock (severity 10 out of 10), and the low access complexity of the attack, suitable for the two hours allotted for the lab. Figure 7.1 shows the lab timeline with the approximate durations of each of its three parts.

[1] *NIST (2014b)*

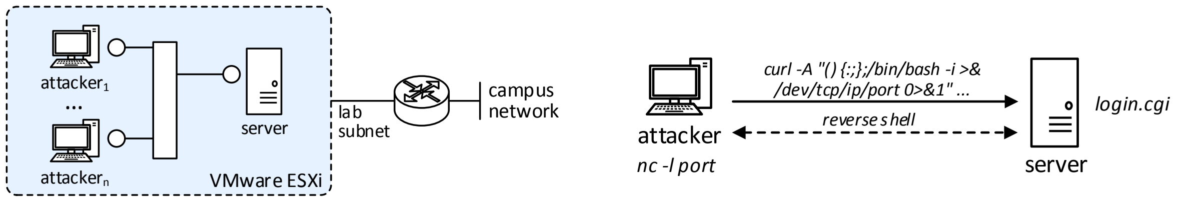
PREPARATION The first part provided an overview of the lab session, followed by a brief introduction to Shellshock. As part of this description, we covered the historical background and relevancy of Bash, and detailed the various attack vectors that can be used to exploit server applications running the vulnerable shell. In addition, we introduced basic background on our particular target server deployment (e.g., Apache, CGI). At the end of this exposition, we ran an interactive demonstration of Shellshock to test students' understanding of the vulnerability and ensure that they were familiarized with the lab environment.

EXPLOITATION CHALLENGE The hands-on part of the lab consisted of a challenge. Students were asked to attack our server and attempt to escalate their privileges after gaining access to the server. To complete this exercise, students needed to build their own exploits and apply the knowledge acquired in the preparation session of the lab. At the end of this exercise, we asked students to fill out an online survey. In order to obtain an unbiased feedback, students were unaware that they were attacking a honey-patched system. We only revealed this information in the third and last part of the lab.

DECEPTION-BASED ACTIVE DEFENSE In the last part of the lab, we first provided a brief overview of deception-based techniques for active defense and offensive countermeasure concepts (e.g., honeypots, decoys, beacons). Then we introduced students to honey-patching and disclosed the fact that our target server was honey-patched, explaining its underlying mechanisms, including misdirection and monitoring capabilities. We also demonstrated the process involved in honey-patching Shellshock. To conclude the exercise, we gave students the opportunity to attack the system once again, for another 30 minutes, and then presented and discussed the monitoring logs generated by the honey-patched system. Before we adjourned, we asked students to fill out a second survey providing feedback about their learning experience.

7.4 LAB DESIGN

From the provisioning of the required physical resources and setup of the lab environment to the preparation of tutorials and challenges, there is a considerable amount of effort involved in organizing a hands-on cyber security lab. Even though the number of students was small, we designed this exercise to scale to a much larger number of participants. In what follows, we highlight some of the preparation steps for this lab.



```
curl -A "() {:;};/bin/bash -i >& /dev/tcp/ip/port 0>&1" ...
nc -l port
```

(a) Lab subnet and virtualization infrastructure (b) Obtaining a reverse shell with Shellshock

Figure 7.2: Lab preparation illustrating (a) lab subnet and virtualization infrastructure and (b) attack demonstration leveraging Shellshock to obtain a reverse shell

7.4.1 *Infrastructure and Preparation*

Figure 7.2a illustrates the infrastructure created for the lab. We built this infrastructure atop VMWare's ESXi, allowing us to quickly and efficiently deploy many linked VMs as needed to create individual guest environments for each participant. The target server and attacker VMs were deployed within the same subnet, and access control rules isolated the lab from the rest of the university network. This created a safe environment in which exploits could be attempted without risk to the surrounding network.

TARGET SERVER The target server was honey-patched against Shellshock and hosted a CGI shell script deployed atop Apache for processing user authentication in a web application specifically created for this lab. To entice students to further exploit the system, decoys were generated with fake user accounts and honey-files containing "interesting" information, such as fake credentials and weakly encrypted user account passwords. To gain escalated access to the decoy, students could discover vulnerable paths concealed within the system. For example, participants might transfer the encrypted password file to their own machines, and crack it with a password cracker (e.g., using a dictionary attack).

MONITORING Decoys also hosted software monitors that collected fine-grained attack information. To minimize the performance impact on decoys, we used two powerful and

Listing 7.3: Decoy's file-system monitoring

```
25/04/2015—13:24:25 /usr/local/apache/cgi—bin/ I_Shocked_You CREATE
25/04/2015—13:24:25 /usr/local/apache/cgi—bin/ I_Shocked_You OPEN
25/04/2015—13:24:25 /usr/local/apache/cgi—bin/ I_Shocked_You ATTRIB
25/04/2015—13:24:25 /usr/local/apache/cgi—bin/ I_Shocked_You CLOSE...
```

Listing 7.4: Decoy's deep inspection of network packets

```
0x0020: 8018 00e5 1aed 0000 0101 080a 0032 9a09  .............2..
0x0030: 0032 9a09 3261 0d0a 495f 5368 6f63 6b65  .2..2a..I_Shocke
0x0040: 645f 596f 750a 6c6f 6769 6e2e 6367 690a  d_You.login.cgi.
0x0050: 6d69 6e65 0a6e 6f5f 796f 755f 6469 646e  mine.no_you_didn
0x0060: 740a 0d0a                                t...
```

highly efficient tools: *inotifywait* (to track modifications made to the file system), and *tcpdump* (to monitor ingress and egress of network packets). To avoid possible tampering with the collected data, all logs were stored outside the decoy environments. In addition, we tuned both monitoring tools to avoid generating spurious outputs (e.g., by excluding certain directories and limiting the monitored network traffic). Listings 7.3 and 7.4 show sample monitoring logs produced after an attack executed by a student. The logged file system events reveal that the student created a file named `I_Shocked_You` in the server's (actually, a decoy's) CGI directory and changed the created file's permissions. In the network logs, we see the response payload returned to the student for an attack that ran the `ls` command on the server.

ATTACKER ENVIRONMENT Each student was assigned a guest VM prepared specifically for the hands-on exercises. Each VM ran Ubuntu 14.04, and came with the minimal tools required to complete the demonstration session (i.e., `curl`, `nc`). Student accounts were configured with administrative privileges, and internet access was not prohibited, allowing easy installation of additional tools as needed. For example, several students downloaded and installed password crackers to use during the hands-on session.

7.4.2 *Interactive Demonstration*

The demonstration delivered at the end of the preparation session consisted of a *no-one-left-behind* exercise, in which the instructor explains each step of the demo and waits until all students have successfully completed it. This strategy worked well given our small group, but would probably need to be adjusted for a larger number of students (e.g., by having more tutors walking around and assisting whoever gets stuck). We used this demo to further clarify concepts introduced in the initial lab presentation and to ensure that all students started the free hands-on session with a basic working knowledge of the techniques used to exploit Shellshock. For instance, Figure 7.2b illustrates one of the attacks we demonstrated, in which the attacker leverages Shellshock to obtain a reverse shell on the vulnerable server.

7.4.3 *Participants*

The lab was open to any student willing to participate, and we did not impose any restrictions on required background. To reach interested students, we announced the lab through the homepages and mailing lists of the security and computer student organizations at UTD. To catch students' attention, we promoted the lab as a hands-on challenge on Shellshock exploitation and defense.

The participants were all CS majors, with limited experience in cyber security (ranging from none to some), with a few who had performed penetration tests before. The lab was staffed by one PhD student and two Masters students who acted as tutors for the lab and offered participants individual assistance as needed. This organization dynamics worked well to solve issues quickly and facilitate the fluidity of the lab.

7.5 SURVEY RESULTS

To informally assess the effectiveness of the lab, we asked students to complete two online surveys (see the appendix) made available through Google Forms. These surveys were anonymous. To minimize the influence of the survey questions on student behavior during the hands-on sessions, the survey questions were not disclosed until after the students completed each portion of the exercise that was surveyed—prior knowledge of the questions would reveal too much about the exercise. Survey questions were phrased as boolean inquiries (1=yes, 0=no) followed by open-ended clarifying questions in which students were given the opportunity to comment.

DECEPTIVENESS OF HONEY-PATCHING The first survey examined the deceptiveness of honey-patching by asking students whether they had realized that they were interacting with a decoy. All students answered *"no"* to this question. From their responses, it is clear that the honey-patched server successfully deceived students for the entire duration of the first hands-on session, which lasted about 30 minutes.

In the last part of the lab, after revealing the deception to students, we asked, "If you were given enough time, what would you attempt to do?" Responses included, *"[I would] look at the services that are running (in the decoy) and try to exploit the honey-patched system."* Another student said, *"[I would] note files of interest and various properties of them (who created them, permissions),"* and another mentioned, *"I would try to find red flags that could be used to probe a honey-patched system."* A particularly noteworthy response was that of a student who said s/he would attempt to relay back to the honey-patch components, in particular the front-end proxy, in order to look for security flaws and exploit them.

These results are a preliminary indication of the efficacy of honey-patching for raising student awareness of cyber deception and counterintelligence gathering, and its educational value for encouraging students to seek deception-exposing strategies and examine exploit outcomes critically rather than accepting them at face value.

LEARNING EXPERIENCE Students also answered general questions about their educational experience. For example, in response to the question, "Did you find this exercise useful for expanding your cyber security education?" students unanimously answered *"yes."* In the open-ended comments, students also said that it was exciting to see how the exploit worked first-hand. Indeed, learning the concepts involved in attacking and defending computer systems in a safe and coherent context seems to entice students' curiosity and develop their interest in applied cyber security.

We also received copious constructive feedback from students on possible ways in which we could improve the lab. These include proposals for new challenges, different methods of attack, and alternative ways to defend against them. Overall, this was a very successful learning experience with a very positive response from students. When asked, "Did this exercise increase your interest in the research side of cyber security?" one student commented, *"I also enjoyed seeing the research being done to take advantage of these kinds of exploits in terms of defense."*

7.6 DISCUSSION AND LESSONS LEARNED

LAB ORGANIZATION We organized this lab combining short, alternating structured (lecturing, demo) and unstructured (free hands-on) sessions. This choice was made to keep students focused and motivated, while giving them freedom to experiment on their own. We believe that this approach helped us to strike a good balance between guided and exploratory learning.

In addition, we believe that concealing the honey-patching deception from students during the first hands-on session raised their interest relative to disclosing it immediately, and was well received by students. While we were initially concerned that students might feel betrayed by instructors once the deception was revealed, our experiences indicate that allowing students to experience a real (but benign and educational) deception during the exercise evokes an element of surprise that students find intriguing and memorable. In particular, we observed a notable increase in interest after we introduced the research on honey-patching and revealed that they had been interacting with decoys since the beginning of the lab. This was evidenced by a surge in questions and discussions.

Second, the delayed reveal opened the way for students to imagine new application scenarios that we did not even cover in our short presentation. For example, a very interesting suggestion was to use honey-patching as a strategy to enhance incidence response and help defenders gather additional attack evidence shortly after a target is compromised.

RESEARCH & KNOWLEDGE TRANSFER Transferring research findings and abstract knowledge into practical use is critical for improving the security posture of cyber space. This includes creating a body of security guidelines, information materials, and more comprehensive education programs focusing on fostering such transition. In our opinion, information assurance and security programs should complement the traditional classroom experience with hands-on exercises in which students are invited to try new research and become armed with state-of-the-art tools and techniques to protect our privacy and the world we live in from emerging cyber threats.

CYBER DECEPTION CTF To cultivate additional student involvement, we also intend to develop a CTF competition at TexSAW with a focus on cyber deception and honey-patching. This will be an offense-defense team challenge, in which participants will learn and practice a variety of skills spanning deception and anti-deception techniques. We envision at least two different ways in which we can organize this competition.

In the first mode, all participants will be taught about honey-patching to use it to misdirect and deceive attacks. In this style of competition, each team will not only try to capture the flag, but also avoid submitting captured decoy flags impersonating genuine flags. To make it more challenging, flag validation and score computation will only occur at the end of each predetermined phase.

A second approach is to enter teams trained in cyber-deceptive active defense techniques into pre-existing CTF competitions, concealing that intended strategy from rival teams. If successful, this could provide empirical evidence of the efficacy of honey-patching and other deceptive defenses for waging cyber warfare. A challenge for such evaluation is finding competitions with rules sufficiently open-ended that they admit these deceptive techniques. Many CTFs are structured such that flag validation is immediate and automatic, making deception less valuable in that context than it is in practice.

7.7 CONCLUSION

Cyber deception is an increasingly important component of effective, real-world cyber defenses. It can be leveraged to level a battlefield that otherwise inherently favors attackers, who succeed if they find just one vulnerability, over defenders, who only succeed if they close all vulnerabilities. By concealing which attacks succeed and which fail, honey-patches give defenders valuable advance intelligence about attacker gambits, and offer opportunities to misdirect attackers away from critical targets toward non-critical targets.

However, like many cyber security paradigms, deception is an arms race. Effective deception depends upon effective skills imparted by effective educational methods. Our initial experiences creating and running active cyber defense lab exercises for computer science students have indicated that honey-patching can be deployed in an educational setting to teach cyber deception in ways that overcome the otherwise predictable (and therefore non-deceptive) classroom environment. We therefore advocate incorporating such exercises into future CTF competitions and into cyber educational curricula to bring these skills to a broader array of upcoming cyber security professionals.

8

RELATED WORK

8.1 DECEPTION IN CYBER SECURITY

8.1.1 *Deception in Warfare*

Deception has been used in warfare since ancient times and has long been recognized as an important tool in intelligence operations.[1] The work traditionally ascribed to the Chinese military strategist Sun Tzu exemplifies this by highlighting deception as key to success in warfare.[2] In ancient Egypt, symbol substitutions were applied to hieroglyphic inscriptions carved into the chambers and tombs of noblemen and pharaohs to obscure their meaning and increase mystery, opening the recorded history of cryptology.[3]

[1] *Latimer (2003)*

[2] *Sun Tzu (1963)*

[3] *Kahn (1974)*

In modern times, deception has developed into a fully fledged doctrine for military operations and strategy.[4] One of the most successful long-term deception campaigns of the 20th century can be traced to the collection of counterespionage documents known as the *Farewell Dossier* created during the Cold War, in which U.S. intelligence teams duped Soviet spies by purposely providing them with faulty specifications of computer programs and hardware, ultimately leading to a major disaster in the trans-Siberian pipeline and causing "inexplicable" problems in Soviet manufacturing and military operations.[5]

[4] *cf., Chairman of the Joint Chiefs of Staff (2012)*

[5] *Rothstein and Whaley (2013)*

Bell and Whaley[6] argue that deception comprises a two-step process entailing simulation and dissimulation, even when only one step is apparent. In this model, the simulation step presents the *false* through mimicking, inventing, and

[6] *Bell and Whaley (1991)*

decoying. Conversely, the dissimulation step involves hiding the *real*, being manifested through masking, repackaging, and dazzling. Alternatively, Dunnigan and Nofi[1] classify deception into concealment, camouflage, false and planted information, lies, displays, ruses, demonstrations, feints, and insights. Each of these groups represents an instance of, or has a direct mapping into one of Bell and Whaley's categories.[2] Traditionally, the Denial & Deception (D&D) literature focuses on such dichotomies between simulation (to reveal fictions) and dissimulation (to conceal facts). Bennett and Waltz[3] refine this taxonomy to include *truth* and *misdirection* as important methods of any successful D&D campaign. Using truth as a foundation to establish the source of target expectations and beliefs, and employing misdirection to manipulate what the target registers, the deceiver is able to induce the target to perceive and accept the revealed facts, while failing to perceive the deceiver's concealed fictions.

[1] *Dunnigan and Nofi (2001)*

[2] *Almeshekah (2015)*

[3] *Bennett and Waltz (2007)*

8.1.2 *Deception Modeling in Cyber Security*

While deception in the context of warfare, diplomacy, and politics has been comprehensively researched and documented over history,[4] until recently efforts devoted to studying deception in the cyber domain were often scarce.[5] Cohen[6] was among the first to propose a model for using deception in computer defenses, outlining a framework describing the basic properties affecting the effectiveness of human and computer deceptions, such as timing and sequence of events, influence of observables, operational security, and resource constraints. His seminal work on the deception toolkit (DTK) introduces a software instrument designed to deceive attackers.

Alternatively, Grazioli and Jarvenpaa[7] propose a classification model for online deception based on masking, dazzling, decoying, mimicking, inventing, relabeling, and double playing, and Rowe[8] introduces a probabilistic model of attacker beliefs for choosing appropriate times and methods to deceive the attacker. Departing from previous works, Rowe and

[4] *Bodmer et al. (2012); Whaley (1969)*

[5] *Heckman et al. (2015)*

[6] *Cohen (1998)*

[7] *Grazioli and Jarvenpaa (2000)*

[8] *Rowe (2004a)*

Rothstein[1] and Rowe[2] use linguistic case theory to provide a cyber deception taxonomy based on the hypothesis that every deception action can be categorized by an associated semantic case or set of cases, including space, time, participant, causality, quality, essence, and speech-act cases. This model pioneered the use of quantitative metrics to support deception planning and to assess the suitability of different deception methods for both offensive and defensive operations in cyberspace.

[1] *Rowe and Rothstein (2004)*

[2] *Rowe (2006)*

Fowler and Nesbit[3] suggest six general principles for effective tactical deception in warfare, which prescribe that deceptions should (1) reinforce enemy expectations, (2) have realistic timing and duration, (3) be integrated with operations, (4) be coordinated with concealment of true intentions, (5) be tailored to contextual requirements, and (6) be imaginative and creative. Rowe and Rothstein[4] translate such principles to the context of cyber warfare involving both offensive and defensive deceptions. These rules highlight limitations of current deception-based defenses. For example, conventional honeypots usually violate the third rule of integration as they are often deployed as *ad hoc*, stand-alone lures isolated from production servers. This makes them easily detectable by most advanced adversaries.

[3] *Fowler and Nesbit (1995)*

[4] *Rowe and Rothstein (2004)*

While these seminal works in cyber deception modeling have opened this new field of research, many subsequent works in cyber deception tools and techniques have lacked a uniform terminology, and have tended to focus on strictly technological characterization of tactics. Such deficiencies have motivated recent research seeking models, taxonomies, and frameworks that can connect the technical plan to the social, behavioral, and cognitive phenomena that emerge from computer deceptions.[5]

[5] *Heckman et al. (2015); Almeshekah (2015)*

Game theoretical models have also been proposed to study cyber deception in cyber security contexts. For instance, Al-Shaer and Rahman[6] propose a game-theoretic framework that uses deception to combat fingerprinting attacks. Pavlovic[7] argues that security is a game of incomplete information where players do not have enough information to predict their opponent's behavior, thus positing that defenders can

[6] *Al-Shaer and Rahman (2015)*

[7] *Pavlovic (2011)*

improve the odds of winning the game by analyzing the behaviors and algorithms of the adversary while obscuring their own. As a practical example, Carroll and Grosu[1] investigate the effects of deploying honeypots on adversarial network security scenarios, and demonstrate that camouflage is an equilibrium strategy for the defender by modeling defender-attacker interaction as a signaling game—a dynamic game of incomplete information. Likewise, La et al.[2] model honeypot-enabled Internet of Things (IoT) networks by looking at a game-theoretic model in which both attackers an defenders try to deceive each other. Garg and Grosu[3] propose a game theoretic framework for modeling deception in honeynets, and study equilibrium solutions to determine the strategies of the attacker and the honeynet system. Alternatively, Crouse[4] introduces an urn-modeling technique for characterizing the probability of success for an attacker in cyber-deception scenarios involving network address shuffling and honeypots.

[1] *Carroll and Grosu (2011)*

[2] *La et al. (2016)*

[3] *Garg and Grosu (2007)*

[4] *Crouse (2012)*

8.1.3 *Deceptive Enhancements to Cyber Security*

Deception mechanisms such as decoys and baits have been used in cyber security for over two decades. In "The Cuckoo's Egg," Stoll[5] documents one of the first public accounts of a successful use of cyber deception for information security and attack attribution. In 1991, AT&T researchers were able to lure and monitor a cracker who had infiltrated their deception "Jail".[6] The deception persisted over several months, affording the defenders enough time to trace the attacker's location and learn his techniques. The success of such cyber-deceptive operations has drawn the attention of the security research community to cyber deception, giving birth to the modern concept of honeypots.[7] Since then, there has been an increasing interest in honeypotting technologies,[8] and a proliferation of other deceptive mechanisms—often labeled with the prefix *honey*—such as honeytokens,[9] honeyfiles,[10] honey encryption,[11] honeywords,[12] and ErsatzPasswords.[13]

[5] *Stoll (1989)*

[6] *Cheswick (1992)*

[7] *Spitzner (2002)*

[8] *Bringer et al. (2012); Spitzner (2003a); Thonnard and Dacier (2008); Provos (2004); Kippo (2009); Glastopf (2009); ThreatStream (2014); Deutsche Telekom AG (2015); Cymmetria (2016)*

[9] *Spitzner (2003b)*

[10] *Yuill et al. (2004)*

[11] *Juels and Ristenpart (2014)*

[12] *Juels and Rivest (2013)*

[13] *Almeshekah et al. (2015)*

Mehresh and Upadhyaya[1] argue that the goal of deception is to influence an adversary's observables by concealing or altering the perceived information. In this sense, some software obfuscation can be seen as deceptive. For example, some obfuscation techniques can be designed to yield wrong disassemblies when analyzed by certain common tools,[2] therefore providing a mechanism to deceive analysts attempting to reverse engineering software. Similarly, Murphy[3] and Murphy et al.[4] discuss the utilization of fingerprint scrubbing and OS obfuscation to conceal from attackers important information and degrade the impact of attacks.

[1] *Mehresh and Upadhyaya (2012)*

[2] *Brand et al. (2010)*

[3] *Murphy (2009)*

[4] *Murphy et al. (2010)*

In digital forensics, machine learning techniques are typically used to determine document authorship, such as the author's identity, demographics, and relational links to other documents.[5] However, such conventional stylometric techniques become ineffective when faced with authors who take measures to hide their identities by obfuscating their writing styles or impersonating other authors.[6] This has motivated a large body of work on detecting stylistic deceptions.[7] The problem of authorship attribution in adversarial settings also has security applications that transcend finding the authorship of rogue documents. For example, malware authorship attribution can be instrumental in helping forensic analysts trace the origins of criminal activities, such as in finding the authors of malware samples[8] and attributing authorship of malicious messages crafted to spam and deceive legitimate users via email.[9]

[5] *Juola (2006); Koppel et al. (2009)*

[6] *Afroz (2013); Afroz et al. (2012)*

[7] *Burgoon and Jr. (2004); Zhou et al. (2004); Warkentin et al. (2010); Pearl and Steyvers (2012); Juola (2012); Feng et al. (2012)*

[8] *Alrabaee et al. (2014); Rosenblum et al. (2011)*

[9] *Alazab et al. (2013)*

In mission-critical operational contexts, when integrated in the prevention and detection layers, deception can help trace attacker motives and strategies, thereby aiding in the development of targeted recovery and adaptation procedures to help critical systems continue delivering essential services despite attacks.[10] Likewise, security mechanisms for cyber-physical systems can employ deception in human computer interactions to make them more resilient against cyber criminals.[11]

[10] *Mehresh and Upadhyaya (2012, 2016)*

[11] *McQueen and Boyer (2009)*

8.1.4 *Offensive Cyber Deception*

Offensively, malware attacks often employ D&D techniques to circumvent detection mechanisms.[1] For example, malvertising campaigns inject malicious advertisements in genuine, vulnerable domains to redirect users to malicious sites that serve malware. Such malvertisements infect their victims by using a multitude of deceptive strategies, such as hiding in plain sight via obfuscated scripts that resemble part of the site's visual architecture, embedding invisible iframes into the webpage, tampering with sponsored links, and infecting content delivery networks.[2] Likewise, *clickjacking* attacks use multiple transparent UI layers to conceal hyperlinks beneath legitimate clickable content, thereby causing unsuspecting victims to perform unintended actions.[3]

[1] *Marpaung et al. (2012)*

[2] *Sood and Enbody (2011); Cova et al. (2010); Mansfield-Devine (2014)*

[3] *Huang et al. (2012)*

Social engineering[4] is another prominent threat class that extensively relies on user deceptions, and includes well-known, yet extremely successful attacks, such as baiting,[5] phishing,[6] and scareware scams.[7]

[4] *Mitnick and Simon (2011)*

[5] *Tischer et al. (2016)*

[6] *Dhamija et al. (2006)*

[7] *Stone-Gross et al. (2013)*

8.1.5 *Deception-based Active Defense*

The same D&D strategies that undermine defenses also facilitate laying traps for adversaries. Gartzke and Lindsay[8] argue that offensive and defensive advantages in cyberspace result from relative organizational capabilities for strategically employing deception. The Stuxnet attack on Iran's nuclear facilities showcases an unprecedented and elaborate use of deceptive techniques at the technological, tactical, and strategic levels in a cyber-sabotage operation that specifically targeted the physical destruction of centrifuges performing uranium enrichment at Natanz.[9] The attack combined a number of ruses, including endpoint protection evasion, self-hiding propagation and replication controls, and the ability to replay pre-recorded, normal control signals to deceive operators.

[8] *Gartzke and Lindsay (2015)*

[9] *Langner (2011)*

Most remarkable, however, was Stuxnet's ability to tactically and strategically coordinate such deceptive, technical capabilities in an operation directed at intelligence gathering and disruption of Iran's nuclear program.[1] In this regard, such offensive deception can be considered *actively defensive*—the perpetrators in the Stuxnet operation are believed to have been trying to proactively defend themselves against a perceived Iranian nuclear threat.[2] Similarly, during the 2007 surge of U.S. combat forces, the NSA has effectively used cyber tools to muddle insurgents' communication devices and deceive enemy forces with fake information, leading them into ambushes prepared by U.S. troops.[3] Alternatively, to protect enterprise networks against cyber attacks, Jones and Laskey[4] present an active defense method based on Bayesian networks that plans and executes cyber deception operations aimed at deterring attackers by making them believe that they are under a cyber attack when in fact they are not.

[1] *Heckman et al. (2015)*

[2] *Sanger (2012)*

[3] *Nakashima (2012)*

[4] *Jones and Laskey (2014)*

Naturally, there are also advantages in being able to detect and counter deceptive efforts. The Line X sabotage[5] is a prime example of successful use of counter-deception for active defense. Although Soviet agents were able to use offensive deception to penetrate Western industry, their actions also made them vulnerable to deception responses and counter-intelligence conducted by the CIA. Rowe[6] describes cyber counter-deception as the use of planned deceptions to defend information systems against attacker deceptions. However, while such *second-order deceptions*[7] remain largely under-utilized in cyber-defensive scenarios,[8] they are frequently used by attackers to search for evidence of honeypots,[9] avoid malware analysis,[10] and conceal their presence and identity on systems into which they trespass.[11] In the virtualization domain, malware attacks often employ stealthy techniques to detect VM environments, within which they behave innocuously and opaquely while being analyzed by antivirus tools.[12]

[5] *Rothstein and Whaley (2013)*

[6] *Rowe (2004b)*

[7] *Rowe (2006)*

[8] *Virvilis et al. (2014); Heckman et al. (2015)*

[9] *McCarty (2003); Krawetz (2004); Rowe et al. (2006); Wang et al. (2010)*

[10] *Brand et al. (2010); You and Yim (2010); Sharif et al. (2008)*

[11] *Jiang et al. (2007); Bahram et al. (2010)*

[12] *Chen et al. (2008)*

This underscores an increasing need for counter-deception mechanisms that are capable of tricking and manipulating advanced attacker deceptions. Heckman et al.[13] discuss a real-time, red team/blue team cyber-wargame experiment that utilized a cyber-deceptive operation in which defenders

[13] *Heckman et al. (2013)*

redirected attackers to a high-interaction honeypot, effectively denying malicious use of the real system while misinforming the adversary with falsified information. Its success should motivate security researchers to examine applications of counter-deception techniques to security defenses. Such techniques should be transparent to users and concealed from adversaries, concurrently limiting attacker gains and increasing the costs of their actions. Promising avenues of research include shielding cyberspace sensors from attackers[1] and exploring inherent asymmetric advantages of using deception in information warfare.[2]

[1] *Rice et al. (2011); Endicott-Popovsky et al. (2009)*

[2] *Sullins (2014); Robertson et al. (2015); Lawson et al. (2011); Whitham (2013, 2014)*

8.2 CYBER AGILITY AND MOVING TARGET DEFENSE

Moving target defenses (MTDs)[3] seek to thwart attacks by mutating or evolving digital environments faster than adversaries can adapt. MTD can be viewed as a subclass of the broader field of *cyber agility*,[4] which includes any reasoned modification to a system or environment in response to a functional, performance, or security need. Such approaches sometimes benefit from deceptive ploys that can impede adversarial adaptation to the defense, but deception is not a requirement for agility or MTD to be effective—if the cyber-maneuver is faster than the enemy can react, the defense can be effective without any deceptive element.

[3] *Jajodia et al. (2011)*

[4] *McDaniel et al. (2014)*

MTD techniques can be broadly classified into *host-based* approaches, such as address space layout randomization,[5] instruction set randomization,[6] multi-variant execution environments,[7] and *network-based* approaches, including address hopping,[8] dynamic routes,[9] and dynamic topology.[10] Using a typical attack kill chain (reconnaissance, access, development, launch, and persistence) as a taxonomy, the primary focus of host-based approaches is on development and launch, while network-based techniques focus primarily on the reconnaissance phase. This is useful for thinking about possible threat models and designing evaluation strategies for each technique

[5] *Kil et al. (2006); Berger and Zorn (2006); Iyer et al. (2010)*

[6] *Onarlioglu et al. (2010)*

[7] *Salamat et al. (2009)*

[8] *Al-Shaer (2011); Carroll et al. (2014); Jafarian et al. (2012)*

[9] *Trassare et al. (2013)*

[10] *Kampanakis et al. (2014)*

(e.g., overhead to perform reconnaissance, and time to map a network topology).

Many of these agile defenses can benefit from deception (e.g., using network decoys to affect the perceived topology of an enterprise network to impede reconnaissance). Dually, good deceptions are often agile (i.e., their performance characteristics are indistinguishable to the target they impersonate), therefore creating a reverse synergy between such technologies. Our work benefits from research advances on MTD and extends the class of possible cyber-maneuvers with a new mechanism based on deception to inform the adaptation process.

8.3 REMOTE EXPLOITATION

Remote, exploitable attacks are one of the biggest threats to IT-security, leading to exposure of sensitive information and high financial losses. While (zero-day) attacks exploiting undisclosed vulnerabilities are the most dangerous, attacks exploiting known vulnerabilities are most prevalent—public disclosure of a vulnerability usually heralds an increase of attacks against it by up to 5 orders of magnitude.[1] Most attacks are remote code injections against vulnerable network applications, and are automatically exploitable by malware without user interaction. Fritz et al.[2] survey the threat landscape of remote code injections and their evolution over the past five years.

[1] *Bilge and Dumitras (2012)*

[2] *Fritz et al. (2013)*

Unfortunately, finding vulnerabilities that lead to remote exploits is becoming easier. ReDeBug[3] finds buggy code that has been copied from project to project. This occurs since programmers often reuse code, and patches are not applied to every version. The Automatic Exploit Generation (AEG) research challenge[4] involves automatically finding vulnerabilities and generating exploits by formalizing the notion of an exploit and analyzing source code. Security patches can also be used to automatically generate exploits, since they reveal details about the underlying vulnerabilities.[5]

[3] *Jang et al. (2012)*

[4] *Avgerinos et al. (2011)*

[5] *Brumley et al. (2008)*

To help overcome this, patch execution consistency models, which guarantee that a patch is safe to apply if the tandem execution of patched and unpatched versions does not diverge, have been recommended as a basis for constructing honeypots that detect and redirect attacks.[1] Our work pursues this goal at the software level, where software exploit detection is easier and more reliable than at the network level. Software diversification has also been proposed as an efficient protection against patch-based attacks.[2]

[1] *Maurer and Brumley (2012)*

[2] *Coppens et al. (2013)*

8.4 HONEYPOTS FOR ATTACK ANALYSIS

Honeypots are information systems resources conceived to attract, detect, and gather attack information. They are designed such that any interaction with a honeypot is likely to be malicious. Although the concept is not new,[3] there has been growing interest in protection and countermeasure mechanisms using honeypots.[4] Honeypots traditionally employ virtualization to trap and investigate attacks.[5] By leveraging VM monitors, honeypots adapt and seamlessly integrate into the network infrastructure,[6] monitoring attacker activities within a compromised system.[7] Nowadays, large *honeyfarms*, supporting on-demand loading of resources, enable large-scale defense scenarios.[8]

Shadow honeypots[9] are a hybrid approach in which a front-end anomaly detection system forwards suspicious requests to a back-end instrumented copy of the target application, which validates the anomaly prediction and improves the anomaly detector's heuristics through feedback. Although the target and instrumented programs may share similar states for detection purposes, shadow honeypots make no effort to deceive attackers into thinking the attack was successful.

In contrast, OpenFire[10] uses a firewall-based approach to forward unwanted messages to decoy machines, making it appear that all ports are open and inducing attackers to target false services. Our work adopts an analogous strategy for software vulnerabilities, making it appear that vulnerabilities are unpatched and inducing attackers to target them.

[3] *Spitzner (2002)*

[4] *Lengyel et al. (2012); Beham et al. (2013); Kulkarni et al. (2012)*

[5] *Provos and Holz (2007); Yegneswaran et al. (2004)*

[6] *Kuwatly et al. (2004)*

[7] *Lengyel et al. (2012); Dagon et al. (2004); Garfinkel and Rosenblum (2003); Beham et al. (2013)*

[8] *Vrable et al. (2005); Jiang et al. (2006)*

[9] *Anagnostakis et al. (2010, 2005)*

[10] *Borders et al. (2007)*

8.5 CLONING FOR SECURITY PURPOSES

Our work benefits from research advances on live cloning,[1] which VM architectures are increasingly using for load balancing and fault-tolerance.[2] In security contexts, VM live cloning can also be used to automate the creation of on-demand honeypots.[3] For instance, dynamic honeypot extraction architectures[4] use a modified version of the Xen hypervisor to detect potential attacks based on analysis of request payload data, and delay their execution until a modified clone of the original system has been created. To fool and distract attackers, sensitive data is removed from the clone's file-system. However, no steps are taken to evade leaking confidential information contained within the cloned process memory image, and the detection strategy is purely system-level, which cannot reliably detect the language-level exploits redirected by honey-patches.

[1] *Sun et al. (2009)*

[2] *Lagar-Cavilla et al. (2009); Zheng et al. (2009)*

[3] *Biedermann et al. (2012); Vrable et al. (2005)*

[4] *Biedermann et al. (2012)*

8.6 DYNAMIC TRACKING OF IN-MEMORY SECRETS

Dynamic taint-tracking lends itself as a natural technique for tracking secrets in software. It has been applied to study sensitive data lifetime (i.e., propagation and duration in memory) in commodity applications,[5] analyze spyware behavior,[6] and impede the propagation of secrets to unauthorized sinks.[7]

TaintBochs[8] uses whole-system simulation to understand secret propagation patterns in several large, widely deployed applications, including Apache, and implements *secure deallocation*[9] to reduce the risk of exposure of in-memory secrets. Panorama[10] builds a system-level information-flow graph using process emulation to identify malicious software tampering with information that was not intended for their consumption. Egele et al.[11] also utilize whole-system dynamic tainting to analyze spyware behavior in web browser components. While valuable, the performance impact of whole-system analyses—often on the order of 2000%[12]—remains a significant

[5] *Chow et al. (2004, 2005)*

[6] *Yin et al. (2007); Egele et al. (2007)*

[7] *Zhu et al. (2011); Enck et al. (2014); Gibler et al. (2012)*

[8] *Chow et al. (2004)*

[9] *Chow et al. (2005)*

[10] *Yin et al. (2007)*

[11] *Egele et al. (2007)*

[12] *Yin et al. (2007); Egele et al. (2007); Chow et al. (2004)*

obstacle, rendering such approaches impractical for most live, high-performance, production server applications.

More recently, there has been growing interest in runtime detection of information leaks.[1] For instance, TaintDroid[2] extends Android's virtualized architecture with taint-tracking support to detect misuses of users' private information across mobile apps. TaintEraser[3] uses dynamic instrumentation to apply taint analysis on binaries for the purpose of identifying and blocking information leaking to restricted output channels. To achieve this, it monitors and rewrites sensitive bytes escaping to the network and the local file system. Our work adopts a different strategy to instrument secret-redaction support into programs, resulting in applications that can proactively respond to attacks by self-censoring their address spaces with minimal overhead.

[1] *Zhu et al. (2011); Enck et al. (2014)*

[2] *Enck et al. (2014)*

[3] *Zhu et al. (2011)*

POINTER TAINTEDNESS In security contexts, many categories of widely exploited, memory-overwrite vulnerabilities (e.g., format string, memory corruption, buffer overflow) have been recognized as detectable by dynamic taint-checking on pointer dereferences.[4] Hookfinder[5] employs data and pointer tainting semantics in a full-system emulation approach to identify malware hooking behaviors in victim systems. Other systems follow a similar technique to capture system-wide information-flow and detect privacy-breaching malware.[6]

With this high practical utility come numerous theoretical and practical challenges for effective pointer tainting.[7] On the theoretical side, there are varied views of how to interpret a pointer's label. (Does it express a property of the pointer value, the values it points to, values read or stored by dereferencing the pointer, or all three?) Different taint tracking application contexts solicit differing interpretations, and the differing interpretations lead to differing taint-tracking methodologies. Our contributions in Chapter 3 include a pointer tainting methodology that is conducive to tracking in-memory secrets.

On the practical side, imprudent pointer tainting often leads to taint explosion in the form of over-tainting or label-creep.[8] This can impair the feasibility of the analysis and

[4] *Chen et al. (2004, 2005); Dalton et al. (2007); Katsunuma et al. (2006); Dalton et al. (2008)*

[5] *Yin et al. (2008)*

[6] *Yin et al. (2007); Egele et al. (2007)*

[7] *Dalton et al. (2010); Slowinska and Bos (2009); Kang et al. (2011)*

[8] *Schwartz et al. (2010); Slowinska and Bos (2009)*

increase the likelihood of crashes in programs that implement data-rewriting policies.[1] To help overcome this, sophisticated strategies involving pointer injection (PI) analysis have been proposed.[2] PI uses a taint bit to track the flow of legitimate pointers and another bit to track the flow of untrusted data, disallowing dereferences of tainted values that do not have a corresponding pointer tainted. Our approach uses static typing information in lieu of PI bits to achieve lower runtime overheads and broader compatibility with low-level legacy code.

[1] *Zhu et al. (2011)*

[2] *Katsunuma et al. (2006); Dalton et al. (2008)*

APPLICATION-LEVEL INSTRUMENTATION Much of the prior work on dynamic taint analysis has employed dynamic binary instrumentation (DBI) frameworks[3] to enforce taint-tracking policies on software. These approaches do not require application recompilation, nor do they depend on source code information. However, despite many optimization advances over the years, dynamic instrumentation still suffers from significant performance overheads (e.g., the $\approx 2000\%$ overheads of the whole-system dynamic taint analyses mentioned above), and therefore cannot support high-performance applications, such as the redaction speeds required for attacker-deceiving honey-patching of production server codes. In contrast, our work benefits from research advances on static-instrumented, dynamic data flow analysis[4] to achieve both high performance and high accuracy by leveraging LLVM's compilation infrastructure to instrument taint-propagating code into server code binaries.

[3] *Newsome and Song (2005); Cheng et al. (2006); Qin et al. (2006); Clause et al. (2007); Zhu et al. (2011); Kemerlis et al. (2012)*

[4] *DFSan (2016); Xu et al. (2006); Lam and Chiueh (2006); Chang et al. (2008)*

8.7 ANOMALY-BASED INTRUSION DETECTION

Anomaly-based IDSes[5] find patterns that do not conform to expected system behavior, and are typically classified into *host-based* and *network-based* approaches.

Host-based anomaly detection recognizes intrusions in the form of anomalous subsequences of system call traces, in which co-occurrence of events is key to characterizing

[5] *cf., Garcia-Teodoro et al. (2009); Modi et al. (2013); Lazarevic et al. (2005); Liao et al. (2013); Chandola et al. (2009); Patcha and Park (2007); Tsai et al. (2009)*

anomalous behavior. For example, malware activity and privilege escalation often manifest anomalous system call patterns.[1] Seminal work in this area has established analogies between the human immune system and intrusion detection through statistical profiling of system events.[2] This has been followed by a number of related approaches using histograms to construct profiles of normal behavior.[3] Another frequently-used approach employs a *sliding window* classifier to map sequences of events into individual output values,[4] converting sequential learning into a classic machine learning problem.

Network-based approaches detect intrusions using network data. Since such systems are typically deployed at the network perimeter, they are designed to find anomalous patterns resulting from attacks launched by outside criminals, such as attempts to disrupt or gain unauthorized access to the network.[5] Network anomaly detection is a very broad research area, and it has been extensively studied in the literature.[6] Major approaches can be grouped into classification-based (e.g., SVM,[7] Bayesian network,[8] neural networks[9]), statistical,[10] and information-theoretic[11] techniques.

Network-based anomaly detection systems can monitor a large number of hosts at relatively low cost, but they are usually opaque to local or encrypted attacks. On the other hand, anomaly detection systems operating at the host level have complete visibility of malicious events, despite encrypted network payloads and obfuscation mechanisms.[12] Our approach therefore complements existing techniques and incorporates host- and network-based features to offer protective capabilities that can resist attacker evasion strategies and detect malicious activity bound to different layers of the software stack.

FEATURE EXTRACTION FOR INTRUSION DETECTION

Various feature extraction and classification techniques have been proposed to perform host- and network-based anomaly detection.[13] Extracting features from encrypted network packets has been intensively studied in the domain of *website fingerprinting*, where attackers attempt to detect the web-

[1] *Chandola et al. (2009)*

[2] *Esponda et al. (2004); Haeseleer et al. (1996); Forrest et al. (1996); Hofmeyr et al. (1998)*

[3] *Cabrera et al. (2001); Marceau (2001)*

[4] *Warrender et al. (1999); Eskin et al. (2001); Cohen (1995)*

[5] *Bhuyan et al. (2014)*

[6] *cf., Hodge and Austin (2004); Bhuyan et al. (2014); Nguyen and Armitage (2008); Ahmed et al. (2016)*

[7] *Eskin et al. (2002)*

[8] *Kruegel et al. (2003)*

[9] *Poojitha et al. (2010)*

[10] *Krügel et al. (2002)*

[11] *Lee and Xiang (2001)*

[12] *Kim et al. (2007)*

[13] *Masud et al. (2011)*

sites visited by their victims. Users typically use anonymous networks, such as Tor, to hide their destination websites.[1] However, by just listening to encrypted packets (i.e., packet headers only), attackers can extract meaningful features and train a classifier to predict destinations. The information present from network packets in each trace is typically summarized to form a histogram feature vector, where the features[2] include packet length and direction. HTML markers, percentage of incoming and outgoing packets, bursts, bandwidth, and website upload time have also been used in addition to pack-length histograms.[3] Packet-word vector approaches additionally leverage natural language processing and vector space models to convert packets to word features for improved classification.[4]

[1] *Wang et al. (2014)*

[2] *Liberatore and Levine (2006)*

[3] *Panchenko et al. (2011); Dyer et al. (2012)*

[4] *Alnaami et al. (2015)*

The Bi-Di system presented in Chapter 4 leverages packet and uni-burst data and introduces bi-directional bursting features for better classification of network streams. On unencrypted data, host-based systems have additionally extracted features by building histograms from co-occurrences and sequences of system events, such as system calls.[5] Our intrusion detection approach uses a hybrid scheme that combines both host- and network-based approaches via a modified ensemble technique.

[5] *Chandola et al. (2009); Cabrera et al. (2001); Marceau (2001)*

9 CONCLUSION

9.1 DISSERTATION SUMMARY

This dissertation advanced language-based software cyber deception as a new discipline of study that integrates deceptive defense capabilities into production software systems. In contrast to most prior approaches, such as honeypotting, the focus here has been on augmenting and re-engineering software systems that offer legitimate services and have genuine production value, rather than merely systems whose primary purpose is to attract attacks. This helps to address threats posed by informed adversaries, such as advanced persistent threat actors and insider threats, who often surgically target only valuable assets and ignore honeypots.

As a flagship example of this new approach, we introduced and formulated honey-patching as a language-level strategy for arming production software with deceptive capabilities that mislead adversaries into wasting time and resources on phantom vulnerabilities and embedded decoy file systems. Honey-patches conceal from attackers the information of which software security vulnerabilities are patched, thereby degrading attackers' methods and disrupting their reconnaissance efforts. They differ from traditional honeypots in that they transparently embed misdirection countermeasures that reside within the actual, mission-critical software systems that attackers are seeking to penetrate, rather than merely implementing independent decoy systems. Thus, honey-patching offers advanced deceptive remediations against informed adversaries who can identify and avoid traditional honeypots.

In order to be adoptable, honey-patches imbue production server software with deceptive capabilities without impairing its performance or intended functionality. These new capabilities make cyber attacks significantly more costly and risky for their perpetrators, and give defenders more time and opportunity to detect and thwart incoming attacks. Our investigations have shown that these advantages have significant applications for intrusion detection, cloud computing security, and moving target defense, and also offer educational value in the form of a new, hands-on approach to teach penetration testing and active cyber defense concepts through honey-patching.

Compiler-assisted secret redaction of program process images is a second, related, example of software cyber deception contributed by the dissertation. Its development introduced a new statically-instrumented semantics of dynamic taint tracking for this purpose, to realize efficient, precise honey-patching of production web servers. This new pointer-combine semantics resists taint over-propagation through graph edges, affording effective dynamic taint-tracking in legacy C codes. Chapter 3 highlights an implementation of the new semantics atop the LLVM compiler infrastructure. The implementation significantly improves the feasibility of dynamic taint-tracking for low-level legacy code that stores secrets in graph data structures. To ease the programmer's annotation burden and avoid taint explosions suffered by prior approaches, it introduces declarative type annotations to C via which software developers can identify confidential process data. Deceptive servers self-redact their address spaces in response to intrusions, affording defenders a new tool for attacker monitoring and disinformation.

Our examination of honey-patching applications for intrusion detection revealed that language-based software cyber deception can ameliorate several longstanding and pernicious challenges for machine learning-based intrusion detection, including scarcity of training data, the high labeling burden for (semi-)supervised learning, and concept differences between honeypot attacks versus those against real victims. Integrating deception-powered application-level feedback into a traditional network-level IDS yields exceptional accuracy, improved agility, and substantially improved data set quality that offers high promise for future IDS research.

Based on these findings, we conclude that cyber-deceptive software engineering approaches to security constitute a potentially high-impact new direction of cyber security research, which offers to substantially increase attacker risk at low cost to defenders.

9.2 DISCUSSION AND FUTURE WORK

9.2.1 *Selective honey-patching*

Our work evaluates the feasibility of honey-patching as realistic application, but raises interesting questions about how to evaluate the strategic advantages or disadvantages of honey-patching various specific vulnerabilities. For example, some patches close vulnerabilities by adding new, legitimate software functionalities. Converting such patches to honey-patches might be inadvisable, since it might treat uses of those new functionalities as attacks. In general, honey-patching should be applied judiciously based on an assessment of attacker and defender risk. Future work should consider how to reliably conduct such assessments. Similarly, honey-patching can be applied selectively to simulate different software versions and achieve versioning consistency.

9.2.2 *Automation*

Our implementation approach offers a semi-manual process for transforming patches into honey-patches. An obvious next step is to automate this by incorporating it into a rewriting tool or compiler. One interesting challenge concerns the question of how to audit or validate the secret redaction step for arbitrary software. Future research should consider facilitating this by applying language-based information flow analyses.[1]

[1] *cf., Sabelfeld and Myers (2003)*

9.2.3 *Active Defense*

Honey-patching enhances the current realm of weaponized software by placing defenders in a favorable position to deploy offensive techniques for reacting to attacks. For example, decoys provide the ideal environment for implementing stealthy traps to disinform attackers and report precisely what attacks are doing in real-time,[1] and further insight into the attackers' *modus operandi* can be gained by forging and acting upon decoy data. There is existing work in this direction in DARPA's Mission-oriented Resilient Clouds (MRC) program.[2]

[1] *Crane et al. (2013)*

[2] *Voris et al. (2012b)*

9.2.4 *Deceptiveness*

The effectiveness of a honey-patch is contingent upon the deceptiveness of decoy environments. Prior work has investigated the problem of how to generate and maintain convincing honey-data for effective attacker deception,[3] but there are other potential avenues of deception discovery that must be considered.

[3] *e.g., Yuill et al. (2006); Bowen et al. (2009); Salem and Stolfo (2011); Voris et al. (2012a)*

Response times are one obvious channel of possible discovery that must be considered. Cloning is efficient but still introduces non-zero response delay for attackers. By collecting enough timing statistics, attackers might try to detect response delays to discern honey-patches. Sections 2.5 and 3.4.3 show that this threat can be mitigated by reducing the performance overhead of cloning to the point where it becomes feasible to artificially delay non-forking requests to obtain a response delay distribution indistinguishable from that of the forking requests. Future work should therefore explore even more techniques for improving cloning performance, particularly in cloud computing contexts (see Chapter 5).

In addition, RedHerring's deceptiveness against discovery through response delays is aided by the plethora of noisy latency sources that most web servers naturally experience, which tend to eclipse the relatively small delays introduced by honey-patching.[4] Unpatched, vulnerable servers often

[4] *Souders (2012)*

respond slower to malicious inputs than to normal traffic,[1] just like honey-patched servers. This suggests that detecting honey-patches by probing for delayed responses to attacks may yield many false positives for attackers. If criminals react to the rise of honey-patching by cautiously avoiding attacks against servers that respond slightly slower when attacked, many otherwise successful attacks will have been thwarted.

[1] *Gadot et al. (2014); Wang et al. (2005)*

Alternatively, attackers who take full control of decoys can potentially read and reverse-engineer the process image's binary code to discover the honey-patch (e.g., by injecting malicious code that reads the process binary and finds the honey-patching implementation). While possible with enough effort, we believe this is nevertheless a significant burden relative to the much easier task of detecting failed exploits against traditionally patched systems. The emergence of artificial software diversity[2] and fine-grained binary randomization tools[3] has made it increasingly difficult to quickly and reliably reverse-engineer arbitrary binary process images. Future work should consider raising the bar further by unloading prominent libraries, such as the honey-patch library, during cloning.

[2] *e.g., Jackson et al. (2011)*

[3] *e.g., Wartell et al. (2012)*

Additionally, RedHerring's decoy environments are constructed to look identical to real distributed web servers from inside the container; for example, many real web servers use LXC containers that look like the decoy LXC containers. Therefore, distinguishing decoys from real web servers on the basis of environmental details (e.g., through `init` process control groups) is difficult for attackers. Although resource exhaustion attacks (e.g., flooding) can cause RedHerring to run out of decoy resources (e.g., containers), this outcome is difficult for attackers to distinguish from running the same attacks against a non-honey-patched server; both result in observationally similar resource exhaustions.

Likewise, real-time behavior of the decoy inevitably differs from the target due to the lack of other, concurrent connections. With a long enough observation period, attackers can reliably detect this difference.[4] We mitigate this by constraining decoy lifetimes with a timeout. This resembles

[4] *cf., Fu et al. (2006)*

unpatched servers that automatically reset when a crash or freeze is detected, and therefore limits the attacker's observations of real-time connection activity without revealing the honey-patch or limiting the attacker's access to decoy data or honeyfiles.

9.2.5 *Detection granularity*

One foundational assumption of our work is that some attacks are not identifiable at the network or system level before they do damage. Thus, detection approaches that monitor network or system logs for malicious activity are not a panacea. For example, encrypted, obfuscated payloads buried in a sea of encrypted connection data, or those that undertake previously unseen malicious behaviors after exploiting known vulnerabilities, might be prohibitively difficult to detect by network or log mining. The goal of our work is to detect such exploits at the software level, and then (1) impede the attack by misdirecting the attacker, (2) lure the attacker to give defenders more time and information to trace and/or prosecute, (3) feed attackers disinformation to lower the effectiveness of current and future attacks, and (4) gather information about attacker gambits to identify and better protect confidential data against future attacks. We believe these goals form a promising template for future work in the discipline.

A

CYBER-DECEPTION LAB SURVEY QUESTIONS

A.1 FIRST SURVEY

Q1. Did you succeed in attacking the server? (yes/no) If yes, what actions did you take after you were able to exploit the vulnerability?
Yes: 7/7, No: 0/7

Q2. Did the vulnerable server raise any red flags? (yes/no)
Yes: 0/7, No: 7/7

Q3. If Yes to Q2: Did you think you were interacting with a real server (i.e., not a trap)? (yes/no) If not, please explain.

Q4. If Yes to Q2: Did you observe anything anomalous in any of the following: file-system, server responses? (yes/no) If yes, how long until you observed them?

A.2 SECOND SURVEY

Q1. After your were told that the system was honey-patched, what actions did you take? Did you try to hack the system? (yes/no)
Yes: 1/7, No: 6/7

Q2. If you were given enough time, what would you attempt to do?

Q3. Did you find this exercise useful for expanding your cyber security education? (yes/no)
Yes: 7/7, No: 0/7

Q4. Were the tutorial instructions clear? (yes/no) If not, please suggest improvements.
Yes: 7/7, No: 0/7

Q5. Were the student instructors helpful and responsive? (yes/no)
Yes: 7/7, No: 0/7

Q6. Did this exercise increase your interest in the research side of cyber security? (yes/no) Please elaborate.
Yes: 7/7, No: 0/7

REFERENCES

Afroz, Sadia. *Deception in Authorship Attribution*. PhD thesis, Drexel University, Philadelphia, Pennsylvania, 2013. *(p. 147)*

Afroz, Sadia, Michael Brennan, and Rachel Greenstadt. Detecting hoaxes, frauds, and deception in writing style online. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, pages 461–475, 2012. *(p. 147)*

Ahmed, Mohiuddin, Abdun Naser Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016. *(pp. 91, 102, and 156)*

Al-Shaer, Ehab. Toward network configuration randomization for moving target defense. In Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and X. Sean Wang, editors, *Moving Target Defense*, pages 153–159. Springer, 2011. *(p. 150)*

Al-Shaer, Ehab and Ashiqur Mohammad Rahman. Attribution, temptation, and expectation: A formal framework for defense-by-deception in cyberwarfare. In Sushil Jajodia, Paulo Shakarian, V.S. Subrahmanian, Vipin Swarup, and Cliff Wang, editors, *Cyber Warfare: Building the Scientific Foundation*, chapter 3, pages 57–80. Springer, 2015. *(p. 145)*

Alazab, Mamoun, Robert Layton, Roderic Broadhurst, and Brigitte Bouhours. Malicious spam emails developments and authorship attribution. In *Proceedings of the 4th IEEE Cybercrime and Trustworthy Computing Workshop (CTC)*, pages 58–68, 2013. *(p. 147)*

Almeshekah, Mohammed H. *Using Deception to Enhance Security: A Taxonomy, Model, and Novel Uses*. PhD thesis, Purdue University, West Lafayette, Indianna, 2015. *(pp. 144 and 145)*

Almeshekah, Mohammed H. and Eugene H. Spafford. Planning and integrating deception into computer security defenses. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 127–138, 2014. *(p. 128)*

(p. 146) Almeshekah, Mohammed H., Christopher N. Gutierrez, Mikhail J. Atallah, and Eugene H. Spafford. ErsatzPasswords: Ending password cracking and detecting password leakage. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, pages 311–320, 2015.

(pp. 87 and 157) Alnaami, Khaled, Gbadebo Ayoade, Asim Siddiqui, Nicholas Ruozzi, Latifur Khan, and Bhavani Thuraisingham. P2V: Effective website fingerprinting using vector space representations. In *Proceedings of the IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 59–66, 2015.

(p. 147) Alrabaee, Saed, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. OBA2: An onion approach to binary code authorship attribution. *Digital Investigation*, 11:S94–S103, 2014.

(p. 152) Anagnostakis, Kostas G., Stelios Sidiroglou, Periklis Akritidis, Konstantinos Xinidis, Evangelos Markatos, and Angelos D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th USENIX Security Symposium*, article 9, 2005.

(p. 152) Anagnostakis, Kostas G., Stelios Sidiroglou, Periklis Akritidis, Michalis Polychronakis, Angelos D. Keromytis, and Evangelos P. Markatos. Shadow honeypots. *International Journal of Computer and Network Security (IJCNS)*, 2(9):1–15, 2010.

(p. 8) Anderson, Ross. Why information security is hard – an economic perspective. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*, pages 358–365, 2001.

(p. 19) Ansel, Jason, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, 2009.

(pp. 19 and 46) Apache. Apache HTTP server project. http://httpd.apache.org, 2014.

(pp. 55 and 129) Araujo, Frederico and Kevin W. Hamlen. Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception. In *Proceedings of the 24th USENIX Security Symposium*, pages 145–159, 2015.

Araujo, Frederico and Kevin W. Hamlen. Embedded honeypotting. In Sushil Jajodia, V.S. Subrahmanian, Vipin Swarup, and Cliff Wang, editors, *Cyber Deception: Building the Scientific Foundation*, chapter 9, pages 195–225. Springer, 2016. *(p. 2)*

Araujo, Frederico, Kevin W. Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 942–953, 2014. *(pp. 14, 20, 65, and 129)*

Araujo, Frederico, Mohammad Shapouri, Sonakshi Pandey, and Kevin Hamlen. Experiences with honey-patching in active cyber security education. In *Proceedings of the 8th Workshop on Cyber Security Experimentation and Test (CSET)*, 2015. *(p. 129)*

Arbaugh, William A., William L. Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33(12):52–59, 2000. *(pp. 5, 111, and 125)*

Attariyan, Mona and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, 2010. *(p. 43)*

Avgerinos, Thanassis, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Proceedings of the 18th Network & Distributed System Security Symposium (NDSS)*, 2011. *(p. 151)*

Axelsson, Stefan. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS)*, pages 1–7, 1999. *(p. 81)*

Bahram, Sina, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. DKSM: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 82–91, 2010. *(p. 149)*

Banerjee, Prith, Richard Friedrich, Cullen Bash, Patrick Goldsack, Bernardo Huberman, John Manley, Chandrakant Patel, *(p. 105)*

Parthasarathy Ranganathan, and Alistair Veitch. Everything as a Service: Powering the new information economy. *IEEE Computer*, 44(3):36–43, 2011.

(p. 43) Bauer, Lujo, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS)*, 2015.

(p. 152) Beham, Michael, Marius Vlad, and Hans P. Reiser. Intrusion detection and honeypots in nested virtualization environments. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–6, 2013.

(p. 143) Bell, John Bowyer and Barton Whaley. *Cheating and Deception*. Transaction Publishers, 1991.

(p. 144) Bennett, Michael and Edward Waltz. *Counterdeception Principles and Applications for National Security*. Artech House, 2007.

(p. 150) Berger, Emery D. and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 158–168, 2006.

(pp. 78, 81, 91, 102, and 156) Bhuyan, Monowar H., Dhruba Kumar Bhattacharyya, and Jugal Kumar Kalita. Network anomaly detection: Methods, systems and tools. *IEEE Communications Surveys & Tutorials*, 16(1): 303–336, 2014.

(p. 153) Biedermann, Sebastian, Martin Mink, and Stefan Katzenbeisser. Fast dynamic extracted honeypots in cloud computing. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, pages 13–18, 2012.

(p. 5) Bifrozt. High interaction honeypot router. https://sourceforge.net/projects/bifrozt, 2014. Accessed May 1, 2016.

(pp. 5 and 151) Bilge, Leyla and Tudor Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 833–844, 2012.

Blum, Avrim L. and Pat Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1): 245–271, 1997. *(p. 80)*

Bodmer, Sean, Max Kilger, Gregory Carpenter, and Jade Jones. *Reverse Deception: Organized Cyber Threat Counter-exploitation*. McGraw-Hill, 2012. *(p. 144)*

Boggs, Nathaniel, Hang Zhao, Senyao Du, and Salvatore J. Stolfo. Synthetic data generation and defense in depth measurement of web applications. In *Proceedings of the 17th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 234–254, 2014. *(pp. 91, 93, and 102)*

Borders, Kevin, Laura Falk, and Atul Prakash. OpenFire: Using deception to reduce network attacks. In *Proceedings of the 3rd International Conference on Security and Privacy in Communications Networks (SecureComm)*, pages 224–233, 2007. *(p. 152)*

Bosman, Erik, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 1–20, 2011. *(p. 43)*

Bowen, Brian M., Shlomo Hershkop, Angelos D. Keromytis, and Salvatore J. Stolfo. Baiting inside attackers using decoy documents. In *Proceedings of the 5th International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 51–70, 2009. *(p. 162)*

Bowers, Kevin D., Ari Juels, and Alina Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 187–198, 2009. *(p. 106)*

Brand, Murray, Craig Valli, and Andrew Woodward. Malware forensics: Discovery of the intent of deception. *Journal of Digital Forensics, Security and Law*, 5(4):31–42, 2010. *(pp. 147 and 149)*

Bringer, Matthew L., Christopher A. Chelmecki, and Hiroshi Fujinoki. A survey: Recent advances and future trends in honeypot research. *International Journal of Computer Network and Information Security*, 4(10), 2012. *(pp. 4 and 146)* *(pp. 111 and 125)* Browne, Hilary K., William A. Arbaugh, John McHugh, and William L. Fithen. A trend analysis of exploitations. In *Proceedings of the 22nd IEEE Symposium on Security & Privacy (S&P)*, pages 214–229, 2001.

(pp. 6, 34, and 151) Brumley, David, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security & Privacy (S&P)*, pages 143–157, 2008.

(p. 106) Burger, Roland A., Christian Cachin, and Elmar Husmann. Cloud, trust, privacy: Trustworthy cloud computing whitepaper. Technical report, TClouds Project, 2013.

(p. 147) Burgoon, Judee K. and Jay F. Nunamaker Jr., editors. *Toward Computer-aided Support for the Detection of Deception*, volume 13(1–2) of *Group Decision and Negotiation*. Springer, 2004.

(p. 127) Burning Glass Technologies. Job market intelligence: Report on the growth of cybersecurity jobs, March 2014.

(pp. 156 and 157) Cabrera, João B.D., Lundy Lewis, and Raman K. Mehra. Detection and classification of intrusions and faults using sequences of system calls. *ACM SIGMOD Record*, 30(4):25–34, 2001.

(p. 91) CAIDA. The cooperative association for Internet data analysis. http://www.caida.org, 2016.

(p. 107) Canonical. Juju application and service modeling tool. https://jujucharms.com, 2016. Accessed May 1, 2016.

(p. 146) Carroll, Thomas E. and Daniel Grosu. A game theoretic investigation of deception in network security. *Security and Communication Networks*, 4(10):1162–1172, 2011.

(p. 150) Carroll, Thomas E., Michael Crouse, Errin W. Fulp, and Kenneth S. Berenhaut. Analysis of network address shuffling as a moving target defense. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 701–706, 2014.

(p. 143) Chairman of the Joint Chiefs of Staff. Joint publication 3-13.4: Military deception, 2012.

Chandola, Varun, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009. *(pp. 78, 81, 100, 101, 155, 156, and 157)*

Chang, Chih-Chung and Chih Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3), 2011. *(p. 90)*

Chang, Walter, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 39–50, 2008. *(pp. 43 and 155)*

Chef. Chef configuration management tool. https://www.chef.io, 2016. Accessed May 1, 2016. *(p. 106)*

Chen, Shuo, Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Formal reasoning of various categories of widely exploited security vulnerabilities by pointer taintedness semantics. In *Proceedings of the IFIP TC11 19th International Conference on Information Security (SEC)*, pages 83–100, 2004. *(p. 154)*

Chen, Shuo, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the 35th International Conference on Dependable Systems and Networks (DSN)*, pages 378–387, 2005. *(p. 154)*

Chen, Xu, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 177–186, 2008. *(p. 149)*

Cheng, Winnie, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC)*, pages 749–754, 2006. *(pp. 43 and 155)*

Cheswick, Bill. An evening with Berferd in which a cracker is lured, endured, and studied. In *Proceedings of the USENIX Winter Technical Conference*, pages 20–24, 1992. *(p. 146)* *(p. 153)* Chow, Jim, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium*, pages 321–336, 2004.

(p. 153) Chow, Jim, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium*, pages 331–346, 2005.

(p. 106) Chow, Richard, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the ACM Workshop on Cloud Computing Security*, pages 85–90, 2009.

(p. 57) Clang. clang: A C language family frontend for LLVM. http://clang.llvm.org, 2016. Accessed May 1, 2016.

(p. 17) Clark, Christopher, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI)*, volume 2, pages 273–286, 2005.

(p. 155) Clause, James, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 196–206, 2007.

(pp. 14 and 93) Codenomicon. The Heartbleed bug. http://heartbleed.com, April 2014.

(p. 144) Cohen, Fred. The deception toolkit. http://www.all.net/dtk, 1998.

(p. 156) Cohen, William W. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, pages 115–123, 1995.

(p. 152) Coppens, Bart, Bjorn De Sutter, and Koen De Bosschere. Protecting your software updates. *IEEE Security & Privacy*, 11(2):47–54, 2013.

(p. 31) Corbet, Jonathan. TCP Connection Repair. http://lwn.net/Articles/495304, 2012.

Cortes, Corinna and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. *(p. 86)*

Cova, Marco, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th ACM International Conference on World Wide Web (WWW)*, pages 281–290, 2010. *(p. 148)*

Cox, Landon P., Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. Spandex: Secure password tracking for Android. In *Proceedings of the 23rd USENIX Security Symposium*, pages 481–494, 2014. *(p. 43)*

Crane, Stephen, Per Larsen, Stefan Brunthaler, and Michael Franz. Booby trapping software. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 95–106, 2013. *(p. 162)*

CRIU. Checkpoint/Restore In Userspace. http://criu.org, 2014. *(pp. 19, 23, and 26)*

Crouse, Michael. *Performance Analysis of Cyber Deception Using Probabilistic Models*. PhD thesis, Wake Forest University, Winston-Salem, North Carolina, 2012. *(p. 146)*

Cymmetria. Cymmetria MazeRunner. Technical report, Cymmetria, 2016. https://www.cymmetria.com/whitepaper. *(p. 146)*

Dagon, David, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian Grizzard, John Levine, and Henry Owen. Honeystat: Local worm detection using honeypots. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 39–58, 2004. *(p. 152)*

Dalton, Michael, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pages 482–493, 2007. *(p. 154)*

Dalton, Michael, Hari Kannan, and Christos Kozyrakis. Real-world buffer overflow protection for userspace & kernelspace. In *Proceedings of the 17th USENIX Security Symposium*, pages 395–410, 2008. *(pp. 154 and 155)*

Dalton, Michael, Hari Kannan, and Christos Kozyrakis. Tainting is not pointless. *ACM/SIGOPS Operating Systems Review (OSR)*, 44 (2):88–92, 2010. *(pp. 49 and 154)* *(pp. 115 and 124)* Darcs. Darcs version control system. http://darcs.net, 2016. Accessed May 1, 2016.

(pp. 77 and 80) Denning, Dorothy E. An intrusion-detection model. *IEEE Transactions on Software Engineering (TSE)*, 13(2):222–232, 1987.

(pp. 5 and 146) Deutsche Telekom AG. T-Pot: DTAG community honeypot project. http://dtag-dev-sec.github.io, 2015. Accessed May 1, 2016.

(pp. 44 and 155) DFSan. Clang DataFlowSanitizer. http://clang.llvm.org/docs/DataFlowSanitizer.html, 2016. Accessed May 1, 2016.

(p. 148) Dhamija, Rachna, J. Doug Tygar, and Marti Hearst. Why phishing works. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 581–590, 2006.

(p. 77) DiMaggio, Jon. The Black Vine cyberespionage group. Symantec Security Response, 2015.

(p. 7) DoD Comptroller. Guidance for performing inventory counts. Technical report, U.S. Office of the Under Secretary of Defense, Financial Improvement and Audit Readiness, August 2015.

(pp. 97 and 99) Dudorov, Dmitry, David Stupples, and Martin Newby. Probability analysis of cyber attack paths against business and commercial enterprise systems. In *Proceedings of the IEEE European Intelligence and Security Informatics Conference (EISIC)*, pages 38–44, 2013.

(p. 19) Duell, Jason. The design and implementation of Berkeley Lab's Linux checkpoint/restart. Technical Report LBNL-54941, University of California at Berkeley, 2002.

(p. 144) Dunnigan, James F. and Albert A. Nofi. *Victory and Deceit: Deception and Trickery at War*. Writers Club Press, 2001.

(pp. 87 and 157) Dyer, Kevin P., Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, pages 332–346, 2012.

(pp. 43, 153, and 154) Egele, Manuel, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 233–246, 2007.

Egele, Manuel, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2): 1–42, 2012. *(p. 43)*

Enck, William, Peter Gilbert, Byung Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM (CACM)*, 57(3):99–106, 2014. *(pp. 43, 153, and 154)*

Endicott-Popovsky, Barbara, Julia Narvaez, Christian Seifert, Deborah A. Frincke, Lori Ross O'Neil, and Chiraag Aval. Use of deception to improve client honeypot detection of drive-by-download attacks. In *Proceedings of the 5th International Conference on Foundations of Augmented Cognition (FAC): Neuroergonomics and Operational Neuroscience*, pages 138–147, 2009. *(p. 150)*

Epigraphic Survey, editor. *Reliefs and Inscriptions at Luxor Temple*, volume 1–2. The Oriental Institute of the University of Chicago, 1994, 1998. *(p. 43)*

Eskin, Eleazar, Wenke Lee, and Salvatore J. Stolfo. Modeling system calls for intrusion detection with dynamic window sizes. In *Proceedings of the DARPA Information Survivability Conference and Exposition II (DISCEX)*, volume 1, pages 165–175, 2001. *(p. 156)*

Eskin, Eleazar, Andrew Arnold, Michael Prerau, Leonid Portnoy, and Salvatore Stolfo. A geometric framework for unsupervised anomaly detection. In *Applications of Data Mining in Computer Security*, pages 77–101. Springer, 2002. *(p. 156)*

Esponda, Fernando, Stephanie Forrest, and Paul Helman. A formal framework for positive and negative detection schemes. *IEEE Transactions on Systems, Man, and Cybernetics (SMC)*, 34(1): 357–373, 2004. *(p. 156)*

Feng, Song, Ritwik Banerjee, and Yejin Choi. Syntactic stylometry for deception detection. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers – Volume 2*, pages 171–175, 2012. *(p. 147)* *(p. 5)* Florian, Cristian. Most vulnerable operating systems and applications in 2014. GFI Software, February 2015.

(p. 156) Forrest, Stephanie, Steven A. Hofmeyr, Aniln Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 17th IEEE Symposium on Security & Privacy (S&P)*, pages 120–128, 1996.

(p. 145) Fowler, Charles A. and Robert F. Nesbit. Tactical deception in air-land warfare. *Journal of Electronic Defense (JED)*, 18(6):37–45, 1995.

(p. 6) Friedman, Gregory H. Evaluation report: The Department of Energy's unclassified cyber security program. Technical Report DOE/IG-0897, U.S. Dept. of Energy, October 2013.

(pp. 5 and 151) Fritz, Jakob, Corrado Leita, and Michalis Polychronakis. Server-side code injection attacks: A historical perspective. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 41–61, 2013.

(p. 163) Fu, Xinwen, Wei Yu, Dan Cheng, Xuejun Tan, and Steve Graham. On recognizing virtual honeypots and countermeasures. In *Proceedings of the 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC)*, pages 211–218, 2006.

(p. 163) Gadot, Ziv, Motty Alon, Lior Rozen, Matan Atad, and Yosefa Shulman Vikalp Shrivastava. Global application & network security report 2013. Technical report, Radware, 2014.

(pp. 78, 81, and 155) Garcia-Teodoro, Pedro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1):18–28, 2009.

(p. 152) Garfinkel, Tal and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Network & Distributed Systems Security Symposium (NDSS)*, pages 191–206, 2003.

(p. 146) Garg, Nandan and Daniel Grosu. Deception in honeynets: A game-theoretic analysis. In *Proceedings of the IEEE Information Assurance and Security Workshop (IAW)*, pages 107–113, 2007.

Gartner. Forecast: Consumer digital storage needs, 2010–2016. http://www.gartner.com/resId=1953315, 2012. Accessed May 1, 2016. *(p. 106)*

Gartzke, Erik and Jon R. Lindsay. Weaving tangled webs: Offense, defense, and deception in cyberspace. *Security Studies*, 24(2): 316–348, 2015. *(p. 148)*

Genuer, Robin, Jean Michel Poggi, and Christine Tuleau-Malot. Variable selection using random forests. *Pattern Recognition Letters*, 31(14):2225–2236, 2010. *(p. 88)*

Gerofi, Balazs, Hajime Fujita, and Yutaka Ishikawa. An efficient process live migration mechanism for load balanced distributed virtual environments. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, pages 197–206, 2010. *(p. 19)*

Gibler, Clint, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST)*, pages 291–307, 2012. *(pp. 43 and 153)*

Glastopf. Web application honeypot. https://github.com/mushorg/glastopf, 2009. Accessed May 1, 2016. *(pp. 5 and 146)*

Google. Protocol Buffers - Google's data interchange format. https://code.google.com/p/protobuf, 2008. *(p. 24)*

Google. Web metrics. https://developers.google.com/speed/articles/web-metrics, 2014. *(p. 38)*

Grazioli, Stefano and Sirkka L. Jarvenpaa. Perils of Internet fraud: An empirical investigation of deception and trust with experienced Internet consumers. *IEEE Transactions on Systems, Man, and Cybernetics (SMC), Part A: Systems and Humans*, 30(4): 395–410, 2000. *(p. 144)*

Gu, A.C. Boxuan, Xinfeng Li, Gang Li, Champion, Zhezhe Chen, Feng Qin, and Dong Xuan. D2Taint: Differentiated and dynamic information flow tracking on smartphones for numerous data sources. In *Proceedings of the 32nd IEEE Conference on Computer Communications (INFOCOM)*, pages 791–799, 2013. *(p. 43)* *(p. 156)* Haeseleer, Patrik D., Stephanie Forrest, and Paul Helman. An immunological approach to change detection: Algorithms, analysis and implications. In *Proceedings of the 17th IEEE Symposium on Security & Privacy (S&P)*, pages 110–119, 1996.

(p. 90) Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11 (1):10–18, 2009.

(p. 101) Haque, Ahsanul, Latifur Khan, and Michael Baron. SAND: Semi-supervised adaptive novel class detection and classification over data stream. In *Proceedings of the 30th Conference on Artificial Intelligence (AAAI)*, pages 1652–1658, 2016.

(pp. 129 and 149) Heckman, Kristin E., Michael J. Walsh, Frank J. Stech, Todd A. O'boyle, Stephen R. DiCato, and Audra F. Herber. Active cyber defense with denial and deception: A cyber-wargame experiment. *Computers & Security*, 37:72–77, 2013.

(pp. 111, 144, 145, and 149) Heckman, Kristin E., Frank J. Stech, Roshan K. Thomas, Ben Schmoker, and Alexander W. Tsow. *Cyber Denial, Deception and Counter Deception: A Framework for Supporting Active Cyber Defense*, volume 64 of *Advances in Information Security*. Sushil Jajodia, editor. Springer, 2015.

(p. 87) Hintz, Andrew. Fingerprinting websites using traffic analysis. In *Proceedings of the 2nd Conference on Privacy Enhancing Technologies (PET)*, pages 171–178, 2003.

(p. 43) Ho, Alex, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 29–41, 2006.

(p. 156) Hodge, Victoria J. and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

(p. 156) Hofmeyr, Steven A., Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

Huang, Lin-Shung, Alex Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *Proceedings of the 21st USENIX Security Symposium*, pages 413–428, 2012. *(p. 148)*

Internet Crime Complaint Center (IC3). 2014 Internet crime report. Federal Bureau of Investigation, May 2015. *(p. 128)*

Iyer, Vivek, Amit Kanitkar, Partha Dasgupta, and Raghunathan Srinivasan. Preventing overflow attacks by memory randomization. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 339–347, 2010. *(p. 150)*

Jackson, Todd, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. Compiler-generated software diversity. In Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and Xiaoyang Sean Wang, editors, *Moving Target Defense – Creating Asymmetric Uncertainty for Cyber Threats*, pages 77–98. Springer, 2011. *(p. 163)*

Jafarian, Jafar Haadi, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: Transparent moving target defense using software defined networking. In *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks (HotSDN)*, pages 127–132, 2012. *(p. 150)*

Jajodia, Sushil, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and X. Sean Wang. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, volume 54. Springer Science & Business Media, 2011. *(p. 150)*

Jang, Jiyong, Abeer Agrawal, and David Brumley. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, pages 48–62, 2012. *(p. 151)*

Jeng, Allen. Minimizing damage from J.P. Morgan's data breach. *InfoSec Reading Room*, 2015. *(p. 77)*

Jiang, Xuxian, Dongyan Xu, and Yi Min Wang. Collapsar: A VM-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. *Journal of Parallel and* *(p. 152)*

Distributed Computing – Special Issue on Security in Grid and Distributed Systems, 66(9):1165–1180, 2006.

(p. 149) Jiang, Xuxian, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 128–138, 2007.

(p. 73) Jim, Trevor, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 275–288, 2002.

(p. 149) Jones, James and Kathryn B. Laskey. Using bayesian attack detection models to drive cyber deception. In *Proceedings of the 11th UAI Bayesian Modeling Applications Workshop (BMAW)*, pages 60–69, 2014.

(pp. 8 and 9) Juels, Ari. A bodyguard of lies: The use of honey objects in information security. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 1–4, 2014.

(p. 146) Juels, Ari and Thomas Ristenpart. Honey encryption: Encryption beyond the brute-force barrier. *IEEE Security & Privacy*, 12(4): 59–62, 2014.

(p. 146) Juels, Ari and Ronald L. Rivest. Honeywords: Making password-cracking detectable. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 145–160, 2013.

(p. 77) Juniper Research. The future of cybercrime and security: Financial and corporate threats and mitigation, 2015.

(p. 147) Juola, Patrick. Authorship attribution. *Foundations and Trends in Information Retrieval*, 1(3):233–334, 2006.

(p. 147) Juola, Patrick. Detecting stylistic deception. In *Proceedings of the Workshop on Computational Approaches to Deception Detection*, pages 91–96, 2012.

(p. 143) Kahn, David. *The Codebreakers*. Weidenfeld and Nicolson, 1974.

Kampanakis, Panos, Harry Perros, and Tsegereda Beyene. SDN-based solutions for moving target defense network protection. In *Proceedings of the IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–6, 2014. *(p. 150)*

Kang, Min Gyung, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS)*, 2011. *(pp. 49 and 154)*

Katsunuma, Satoshi, Hiroyuki Kurita, Ryota Shioya, Kazuto Shimizu, Hidetsugu Irie, Mashiro Goshima, and Shuichi Sakai. Base address recognition with data flow tracking for injection attack detection. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 165–172, 2006. *(pp. 154 and 155)*

Kemerlis, Vasileios P., Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th Conference on Virtual Execution Environments (VEE)*, pages 121–132, 2012. *(p. 155)*

Kerckhoffs, Auguste. La cryptographie militaire. *Journal Sciences Militaires*, IX:5–38, 1883. *(p. 10)*

Khan, Safwan M. and Kevin W. Hamlen. AnonymousCloud: A data ownership privacy provider framework in cloud computing. In *Proceedings of the 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 170–176, 2012a. *(p. 106)*

Khan, Safwan M. and Kevin W. Hamlen. Computation certification as a service in the cloud. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 434–441, 2013. *(p. 106)*

Khan, Safwan Mahmud and Kevin W Hamlen. Hatman: Intra-cloud trust management for Hadoop. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)*, pages 494–501, 2012b. *(p. 106)* *(p. 106)* Khan, Safwan Mahmud, Kevin W. Hamlen, and Murat Kantarcioglu. Silver lining: Enforcing secure information flow at the cloud edge. In *Proceedings of the 2nd IEEE International Conference on Cloud Engineering (IC2E)*, pages 37–46, 2014.

(p. 150) Kil, Chongkyung, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 339–348, 2006.

(p. 156) Kim, Jungwon, Peter J. Bentley, Uwe Aickelin, Julie Greensmith, Gianni Tedesco, and Jamie Twycross. Immune system approaches to intrusion detection—a review. *Natural Computing*, 6(4): 413–466, 2007.

(pp. 5 and 146) Kippo. SSH honeypot. https://github.com/desaster/kippo, 2009. Accessed May 1, 2016.

(p. 147) Koppel, Moshe, Jonathan Schler, and Shlomo Argamon. Computational methods in authorship attribution. *Journal of the American Society for Information Science and Technology (JASIST)*, 60(1): 9–26, 2009.

(p. 149) Krawetz, Neal. Anti-honeypot technology. *IEEE Security & Privacy*, 2(1):76–79, 2004.

(p. 156) Kruegel, Christopher, Darren Mutz, William Robertson, and Fredrik Valeur. Bayesian event classification for intrusion detection. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, pages 14–23, 2003.

(p. 156) Krügel, Christopher, Thomas Toth, and Engin Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the 17th ACM Symposium on Applied Computing (SAC)*, pages 201–208, 2002.

(p. 152) Kulkarni, Saurabh, Madhumitra Mutalik, Prathamesh Kulkarni, and Tarun Gupta. Honeydoop – a system for on-demand virtual high interaction honeypots. In *Proceedings of the International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 743–747, 2012.

Kuwatly, Iyad, Malek Sraj, Zaid Al Masri, and Hassan Artail. A dynamic honeypot design for intrusion detection. In *Proceedings of the IEEE/ACS International Conference on Pervasive Services (ICPS)*, pages 95–104, 2004. *(p. 152)*

La, Quang Duy, Tony Quek, Jemin Lee, Shi Jin, and Hongbo Zhu. Deceptive attack and defense game in honeypot-enabled networks for the internet of things. *IEEE Internet of Things Journal*, 2016. *(p. 146)*

Lagar-Cavilla, Horacio Andrés, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, pages 1–12, 2009. *(p. 153)*

Lam, Lap Chung and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 463–472, 2006. *(p. 155)*

Langner, Ralph. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011. *(pp. 128 and 148)*

Latimer, Jon. *Deception in War: The Art of the Bluff, the Value of Deceit, and the Most Thrilling Episodes of Cunning in Military History, From the Trojan Horse to the Gulf War.* Overlook Books, 2003. *(p. 143)*

Lattner, Chris and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*, pages 75–88, 2004. *(p. 44)*

Lawson, Jamie, Rajdeep Singh, Michael Hultner, and Kartik B. Ariyur. Deception robust control for automated cyber defense resource allocation. In *Proceedings of the IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA)*, pages 56–59, 2011. *(p. 150)*

Lazarevic, Aleksandar, Vipin Kumar, and Jaideep Srivastava. *(p. 155)*

Intrusion detection: A survey. In *Managing Cyber Threats*, pages 19–78. Springer, 2005.

(pp. 80 and 156) Lee, Wenke and Dong Xiang. Information-theoretic measures for anomaly detection. In *Proceedings of the 22nd IEEE Symposium on Security & Privacy (S&P)*, pages 130–143, 2001.

(p. 152) Lengyel, Tamas K., Justin Neumann, Steve Maresca, Bryan D. Payne, and Aggelos Kiayias. Virtual machine introspection in a hybrid honeypot architecture. In *Proceedings of the 5th USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2012.

(p. 155) Liao, Hung-Jen, Chun Hung Richard Lin, Ying Chih Lin, and Kuang Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

(p. 157) Liberatore, Marc and Brian Neil Levine. Inferring the source of encrypted HTTP connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 255–263, 2006.

(p. 40) Lighttpd. Lighttpd server project. http://www.lighttpd.net, 2014.

(p. 128) Lingenheld, Michael. The unfortunate growth sector: Cybersecurity. *Forbes*, April 2015.

(p. 111) Lippmann, Richard, Seth Webster, and Douglas Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 307–326, 2002.

(p. 128) Luo, Xin, Richard Brody, Alessandro Seazzu, and Stephen Burd. Social engineering: The neglected human factor for information security management. *Information Resources Management Journal (IRMJ)*, 24(3):1–8, 2011.

(pp. 20 and 83) LXC. Linux containers. http://linuxcontainers.org, 2014. Accessed May 1, 2016.

(p. 115) Lynagh, Ian. An algebra of patches. http://urchin.earth.li/~ian/conflictors/paper-2006-10-30.pdf, 2006.

Manandhar, Prajowal and Zeyar Aung. Towards practical anomaly-based intrusion detection by outlier mining on TCP packets. In *Proceedings of the 25th International Conference on Database and Expert Systems Applications (DEXA)*, pages 164–173, 2014. *(p. 102)*

Mansfield-Devine, Steve. The dark side of advertising. *Computer Fraud & Security*, 2014(11):5–8, 2014. *(p. 148)*

Marceau, Carla. Characterizing the behavior of a program using multiple-length n-grams. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 101–110, 2001. *(pp. 156 and 157)*

Marpaung, Jonathan A.P., Mangal Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: State of the art and challenges. In *Proceedings of the 14th IEEE International Conference on Advanced Communication Technology (ICACT)*, pages 744–749, 2012. *(p. 148)*

Masud, Mehedy, Latifur Khan, and Bhavani Thuraisingham. *Data Mining Tools for Malware Detection*. CRC Press, 2011. *(p. 156)*

Maurer, Matthew and David Brumley. Tachyon: Tandem execution for efficient live patch testing. In *Proceedings of the 21st USENIX Security Symposium*, pages 617–630, 2012. *(p. 152)*

McCarty, Bill. The honeynet arms race. *IEEE Security & Privacy*, 1 (6):79–82, 2003. *(p. 149)*

McDaniel, Patrick, Trent Jaeger, Thomas F. La Porta, Nicolas Papernot, Robert J. Walls, Alexander Kott, Lisa Marvel, Ananthram Swami, Prasant Mohapatra, Srikanth V. Krishnamurthy, and Iulian Neamtiu. Security and science of agility. In *Proceedings of the 1st ACM Workshop on Moving Target Defense (MTD)*, pages 13–19, 2014. *(p. 150)*

McQueen, Miles A. and Wayne F. Boyer. Deception used for cyber defense of control systems. In *Proceedings of the 2nd Conference on Human System Interactions (HSI)*, pages 621–628, 2009. *(p. 147)*

Mehresh, Ruchika and Shambhu Upadhyaya. A deception framework for survivability against next generation cyber attacks. In *Proceedings of the 11th International Conference on Security and Management (SAM)*, pages 1–7, 2012. *(p. 147)* *(p. 147)* Mehresh, Ruchika and Shambhu J. Upadhyaya. Deception-based survivability. In Chip Hong Chang and Miodrag Potkonjak, editors, *Secure System Design and Trustable Computing*, pages 521–537. Springer, 2016.

(p. 8) Merkow, Mark S. and Jim Breithaupt. *Information Security: Principles and Practices.* Pearson Education, 2014.

(p. 17) Milojičić, Dejan S., Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

(p. 128) Mink, Martin and Felix C. Freiling. Is attack better than defense? teaching information security the right way. In *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development (InfoSecCD)*, pages 44–48, 2006.

(p. 148) Mitnick, Kevin D. and William L. Simon. *The Art of Deception: Controlling the Human Element of Security*. John Wiley & Sons, 2011.

(p. 155) Modi, Chirag, Dhiren Patel, Bhavesh Borisaniya, Hiren Patel, Avi Patel, and Muttukrishnan Rajarajan. A survey of intrusion detection techniques in cloud. *Journal of Network and Computer Applications*, 36(1):42–57, 2013.

(p. 147) Murphy, Sherry, Todd McDonald, and Robert Mills. An application of deception in cyberspace: Operating system obfuscation. In *Proceedings of the 5th International Conference on Information Warfare and Security (ICIW)*, pages 241–249, 2010.

(p. 147) Murphy, Sherry B. Deceiving adversary network scanning efforts using host-based deception. Technical Report ADA502233, Air Force Institute of Technology, Wright-patterson Air Force Base, 2009.

(p. 149) Nakashima, Ellen. U.S. accelerating cyberweapon research. *The Washington Post*, March 2012.

(p. 106) Nepal, Surya, Shiping Chen, Jinhui Yao, and Danan Thilakanathan. DIaaS: Data integrity as a service in the cloud. In *Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD)*, pages 308–315, 2011.

Netcraft. Are there really lots of vulnerable Apache web servers? http://news.netcraft.com/archives/2014/02/07, 2014. *(p. 19)*

Netcraft. Web server survey. http://news.netcraft.com/archives/category/web-server-survey, January 2015. *(p. 65)*

Newsome, James and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network & Distributed System Security Symposium (NDSS)*, 2005. *(pp. 43 and 155)*

Nginx. Nginx server project. http://nginx.org, 2014. *(p. 40)*

Nguyen, Thuy T.T. and Grenville Armitage. A survey of techniques for Internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008. *(p. 156)*

Nguyen-tuong, Anh, Salvatore Guarnieri, Doug Greene, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the IFIP TC11 20th International Conference on Information Security (SEC)*, pages 372–382, 2005. *(p. 43)*

NIST. Vulnerability summary for CVE-2014-6277. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6277, October 2014a. *(p. 113)*

NIST. The Shellshock Bash vulnerability. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271, September 2014b. *(pp. 93, 130, and 132)*

Novetta Threat Research Group. Operation Blockbuster: Unraveling the long thread of the Sony attack, 2016. *(p. 77)*

Ohloh. Apache HTTP server statistics. http://www.ohloh.net/p/apache, 2014. *(pp. 19 and 65)*

Onarlioglu, Kaan, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, pages 49–58, 2010. *(p. 150)*

Pai, Vivek S., Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the Conference on USENIX Annual Technical Conference (ATC)*, article 15, 1999. *(p. 40)* *(pp. 87 and 157)* Panchenko, Andriy, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 103–114, 2011.

(p. 43) Papagiannis, Ioannis, Matteo Migliavacca, and Peter Pietzuch. PHP Aspis: Using partial taint tracking to protect against injection attacks. In *Proceedings of the 2nd USENIX Conference on Web Application Development (WebApps)*, article 2, 2011.

(pp. 78, 81, and 155) Patcha, Animesh and Jung Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

(p. 128) Patriciu, Victor-Valeriu and Adrian Constantin Furtuna. Guide for designing cyber security exercises. In *Proceedings of the 8th WSEAS International Conference on Recent Advances in E-Activities, Information Security and Privacy (E-ACTIVITIES)*, pages 172–177, 2009.

(p. 145) Pavlovic, Dusko. Gaming security by obscurity. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 125–140, 2011.

(p. 147) Pearl, Lisa and Mark Steyvers. Detecting authorship deception: A supervised machine learning approach using author writeprints. *Literary and Linguistic Computing*, 27(2):183–196, 2012.

(p. 106) Pearson, Siani. Taking account of privacy when designing cloud computing services. In *Proceedings of the ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 44–52. IEEE, 2009.

(p. 90) Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011.

(p. 2) Pingree, Lawrence. Emerging technology analysis: Deception techniques and technologies create security technology business opportunities. *Gartner*, July 2015. ID:G00278434.

Platt, John C. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pages 61–74. MIT Press, 1999. *(p. 89)*

Poojitha, G., K. Nandha Kumar, and P. Jayarami Reddy. Intrusion detection using artificial neural network. In *Proceedings of the IEEE International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–7, 2010. *(p. 156)*

Portokalidis, Georgios, Asia Slowinska, and Herbert Bos. Argos: An emulator for fingerprinting zero-day attacks. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 15–27, 2006. *(p. 43)*

Provos, Niels. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, pages 1–14, 2004. *(pp. 5 and 146)*

Provos, Niels and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection.* Addison-Wesley, 2007. *(p. 152)*

Puppet. Puppet configuration management tool. https://www.puppet.com, 2016. Accessed May 1, 2016. *(p. 106)*

Py-idstools. Py-idstools Python library. https://github.com/jasonish/py-idstools, 2016. Accessed May 1, 2016. *(p. 124)*

Qin, Feng, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*, pages 135–148, 2006. *(p. 155)*

Rescorla, Eric. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, 2005. *(pp. 111 and 125)*

Rice, Mason, Daniel Guernsey, and Sujeet Shenoi. Using deception to shield cyberspace sensors. In *Proceedings of the 5th IFIP WG 11.10 International Conference on Critical Infrastructure Protection (ICCIP)*, pages 3–18, 2011. *(p. 150)*

Robertson, Seth, Scott Alexander, Josephine Micallef, Jonathan Pucci, James Tanis, and Anthony Macera. CINDAM: Customized information networks for deception and attack mitigation. In *Proceedings of the IEEE 9th International Conference on* *(p. 150)*

Self-Adaptive and Self-Organizing Systems Workshop (SASOW), pages 114–119, 2015.

(p. 147) Rosenblum, Nathan, Xiaojin Zhu, and Barton P. Miller. Who wrote this code? identifying the authors of program binaries. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS)*, pages 172–189, 2011.

(pp. 143 and 149) Rothstein, Hy and Barton Whaley. *The Art and Science of Military Deception*. Artech House, 2013.

(p. 114) Roundy, David. Darcs: Distributed version management in Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 1–4, 2005.

(p. 144) Rowe, Neil C. Designing good deceptions in defense of information systems. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, pages 418–427, 2004a.

(p. 149) Rowe, Neil C. A model of deception during cyber-attacks on information systems. In *Proceedings of the IEEE 1st Symposium on Multi-Agent Security and Survivability*, pages 21–30, 2004b.

(pp. 145 and 149) Rowe, Neil C. A taxonomy of deception in cyberspace. In *Proceedings of the 1st International Conference on Information Warfare and Security (ICCWS)*, 2006.

(pp. 144 and 145) Rowe, Neil C. and Hy S. Rothstein. Two taxonomies of deception for attacks on information systems. *Journal of Information Warfare*, 3(2):27–39, 2004.

(p. 149) Rowe, Neil C., Binh T. Duong, and E. John Custy. Fake honeypots: A defensive tactic for cyberspace. In *Proceedings of the IEEE Information Assurance Workshop (IAW)*, pages 223–230, 2006.

(pp. 74 and 161) Sabelfeld, Andrei and Andrew C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

(p. 77) Sager, Tony. Killing advanced threats in their tracks: An intelligent approach to attack prevention. *InfoSec Reading Room*, 2014.

(p. 150) Salamat, Babak, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and

monitoring of program variants in user-space. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, pages 33–46, 2009.

Salem, Malek Ben and Salvatore J. Stolfo. Decoy document deployment for effective masquerade attack detection. In *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 35–54, 2011. *(p. 162)*

Sampson, Adrian. Quala: Type qualifiers for LLVM/Clang. https://github.com/sampsyo/quala, 2014. Accessed May 1, 2016. *(p. 57)*

Sanger, David E. Obama order sped up wave of cyberattacks against Iran. *The New York Times*, June 2012. *(p. 149)*

Santos, Nuno, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. *Proceedings of the USENIX Workshop in Hot Topics in Cloud Computing (HotCloud)*, 2009. *(p. 106)*

Schneider, Fred B., J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics: 10 Years Back, 10 Years Ahead*, pages 86–101. Springer, 2001. *(p. 7)*

Schwartz, Edward J., Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P)*, pages 317–331, 2010. *(pp. 43, 49, 50, and 154)*

Selenium. Selenium browser automation. http://www.seleniumhq.org, 2016. *(p. 92)*

Sequeira, Karlton and Mohammed Zaki. ADMIT: Anomaly-based data mining for intrusions. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 386–395, 2002. *(p. 80)*

Serebryany, Konstantin, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 309–318, 2012. *(p. 70)* *(p. 43)* Sezer, Emre Can, Peng Ning, Chongkyung Kil, and Jun Xu. MemSherlock: An automated debugger for unknown memory corruption vulnerabilities. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 562–572, 2007.

(p. 5) Shadowd. Shadow daemon Web application firewall. http://dtag-dev-sec.github.io, 2015. Accessed May 1, 2016.

(p. 149) Sharif, Monirul I., Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 15th Annual Network & Distributed System Security Symposium (NDSS)*, 2008.

(p. 80) Sinclair, Chris, Lyn Pierce, and Sara Matzner. An application of machine learning to network intrusion detection. In *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC)*, pages 371–377, 1999.

(pp. 49 and 154) Slowinska, Asia and Herbert Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 61–74, 2009.

(pp. 78, 80, and 102) Sommer, Robin and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P)*, pages 305–316, 2010.

(p. 148) Sood, Aditya K. and Richard J. Enbody. Malvertising—exploiting Web advertising. *Computer Fraud & Security*, 2011(4):11–16, 2011.

(pp. 71 and 99) Souders, Steve. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly, 2007.

(p. 162) Souders, Steve. The performance golden rule. http://www.stevesouders.com/blog/2012/02/10/the-performance-golden-rule, February 2012.

(pp. 4, 146, and 152) Spitzner, Lance. *Honeypots: Tracking Hackers*. Addison-Wesley, 2002.

(pp. 2 and 146) Spitzner, Lance. The Honeynet project: Trapping the hackers. *IEEE Security & Privacy*, 1(2):15–23, 2003a.

Spitzner, Lance. Honeytokens: The other honeypot. *Symenatec Connect*, 2003b. http://www.symantec.com/connect/articles/honeytokens-other-honeypot. *(p. 146)*

Stoll, Cliord P. *The Cuckoo's Egg: Tracing a Spy Through the Maze of Computer Espionage*. Doubleday, 1989. *(p. 146)*

Stone-Gross, Brett, Ryan Abman, Richard A. Kemmerer, Christopher Kruegel, Douglas G. Steigerwald, and Giovanni Vigna. The underground economy of fake antivirus software. In Bruce Schneier, editor, *Economics of Information Security and Privacy III*, pages 55–78. Springer, 2013. *(p. 148)*

Suh, G. Edward, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, 2004. *(p. 43)*

Sullins, John P. Deception and virtue in robotic and cyber warfare. In Luciano Floridi and Mariarosaria Taddeo, editors, *The Ethics of Information Warfare*, pages 187–201. Springer, 2014. *(p. 150)*

Sun, Yifeng, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, Binbin Zhang, Haogang Chen, and Xiaoming Li. Fast live cloning of virtual machine based on Xen. In *Proceedings of the 11th IEEE Conference on High Performance Computing and Communications (HPCC)*, pages 392–399, 2009. *(p. 153)*

Sun Tzu. *The Art of War, Translated by Samuel B. Griffith*. Oxford University Press, 1963. *(p. 143)*

Symantec. Internet security threat report, vol. 21, April 2016. *(p. 77)*

Takabi, Hassan, James B.D. Joshi, and Gail Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 8(6):24–31, 2010. *(p. 106)*

The Economic Times. New technique Red Herring fights 'Heartbleed' virus. *The Times of India*, April 15, 2014. *(p. 16)*

Thonnard, Olivier and Marc Dacier. A framework for attack patterns' discovery in honeynet data. *Digital Investigation: The International Journal of Digital Forensics & Incident Response* *(pp. 2 and 146)* *(the Proceedings of the 8th Annual Digital Forensics Research Conference (DFRW))*, 5:S128–S139, 2008.

(pp. 5 and 146) ThreatStream. Modern honey network. https://threatstream.github.io/mhn, 2014. Accessed May 1, 2016.

(p. 148) Tischer, Matthew, Zakir Durumeric, Sam Foster, Sunny Duan, Alec Mori, Elie Bursztein, and Michael Bailey. Users really do plug in USB drives they find. In *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P)*, pages 306–319, 2016.

(pp. 119 and 124) Toolswatch. vFeed – The correlated vulnerability and threat database. https://github.com/toolswatch/vFeed, 2016. Accessed May 1, 2016.

(p. 150) Trassare, Samuel T., Robert Beverly, and David Alderson. A technique for network topology deception. In *Proceedings of the IEEE Military Communications Conference (MILCOM)*, pages 1795–1800, 2013.

(p. 155) Tsai, Chih-Fong, Yu Feng Hsu, Chia Ying Lin, and Wei Yang Lin. Intrusion detection by machine learning: A review. *Expert Systems with Applications*, 36(10):11994–12000, 2009.

(p. 128) Twitchell, Douglas P. Social engineering in information assurance curricula. In *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development (InfoSecCD)*, pages 191–193, 2006.

(p. 128) U.S. Air Force Materiel Command. Capabilities for cyber resiliency. Broad Agency Announcement, Solicitation BAA-RIK-14-07, August 2014.

(p. 78) Vasilomanolakis, Emmanouil, Shankar Karuppayah, Max Mühlhäuser, and Mathias Fischer. Taxonomy and survey of collaborative intrusion detection. *ACM Computing Surveys*, 47 (4), 2015.

(p. 128) Vigna, Giovanni. Teaching network security through live exercises. In Cynthia Irvine and Helen Armstrong, editors, *Security Education and Critical Infrastructures*, pages 3–18. Kluwer Academic Publishers, 2003.

Virvilis, Nikos, Bart Vanautgaerden, and Oscar Serrano Serrano. Changing the game: The art of deceiving sophisticated attackers. In *Proceedings of the 6th IEEE International Conference on Cyber Conflict (CyCon)*, pages 87–97, 2014. *(p. 149)*

Voris, Jonathan, Nathaniel Boggs, and Salvatore J. Stolfo. Lost in translation: Improving decoy documents via automated translation. In *Proceedings of the IEEE Symposium on Security & Privacy Workshops (S&PW)*, pages 129–133, 2012a. *(p. 162)*

Voris, Jonathan, Jill Jermyn, Angelos D. Keromytis, and Salvatore J. Stolfo. Bait and snitch: Defending computer systems with decoys. In *Proceedings of the 3rd Conference on Cyber Infrastructure Protection (CIP)*, 2012b. *(p. 162)*

Voris, Jonathan, Jill Jermyn, Nathaniel Boggs, and Salvatore Stolfo. Fox in the trap: Thwarting masqueraders via automated decoy document deployment. In *Proceedings of the 8th ACM SIGOPS European Workshop on System Security (EuroSec)*, article 3, 2015. *(p. 3)*

Vrable, Michael, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 148–162, 2005. *(pp. 152 and 153)*

Wang, Chao, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in HPC environments. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008. *(p. 19)*

Wang, Ju, Xin Liu, and Andrew A. Chien. Empirical study of tolerating denial-of-service attacks with a proxy network. In *Proceedings of the 14th USENIX Security Symposium*, pages 51–64, 2005. *(p. 163)*

Wang, Ping, Lei Wu, Ryan Cunningham, and Cliff C. Zou. Honeypot detection in advanced botnet attacks. *International Journal of Information and Computer Security (IJICS)*, 4(1):30–51, 2010. *(p. 149)*

Wang, Tao, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the 23rd USENIX Security Symposium*, pages 143–157, 2014. *(pp. 87 and 157)* *(p. 147)* Warkentin, Darcy, Michael Woodworth, Jeffrey T. Hancock, and Nicole Cormier. Warrants and deception in computer mediated communication. In *Proceedings of the 22nd ACM Conference on Computer Supported Cooperative Work (CSCW)*, pages 9–12, 2010.

(p. 156) Warrender, Christina, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 20th IEEE Symposium on Security & Privacy (S&P)*, pages 133–145, 1999.

(p. 163) Wartell, Richard, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, 2012.

(p. 144) Whaley, Barton. Stratagem: Deception and surprise in war. Technical report, Center for International Studies, Massachusetts Institute of Technology, 1969.

(p. 17) Whitaker, Andrew, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 169–182, 2004.

(p. 150) Whitham, Ben. Canary files: Generating fake files to detect critical data loss from complex computer networks. In *Proceedings of the 2nd International Conference on Cyber Security, Cyber Peacefare and Digital Forensic (CyberSec)*, pages 170–179, 2013.

(p. 150) Whitham, Ben. Design requirements for generating deceptive content to protect document repositories. In *Proceedings of the 15th Australian Information Warfare Conference*, pages 20–30, 2014.

(pp. 43 and 155) Xu, Wei, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, article 9, 2006.

(p. 152) Yegneswaran, Vinod, Paul Barford, and Dave Plonka. On the design and use of Internet sinks for network abuse monitoring.

In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 146–165, 2004.

Yin, Heng, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 116–127, 2007. *(pp. 43, 153, and 154)*

Yin, Heng, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network & Distributed System Security Symposium (NDSS)*, 2008. *(p. 154)*

You, Ilsun and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *Proceedings of the 5th IEEE International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA)*, pages 297–300, 2010. *(p. 149)*

Yuill, Jim, Mike Zappe, Dorothy Denning, and Fred Feer. Honeyfiles: Deceptive files for intrusion detection. In *Proceedings of the 5th IEEE International Workshop on Information Assurance (IWIA)*, pages 116–122, 2004. *(pp. 3 and 146)*

Yuill, Jim, Dorothy Denning, and Fred Feer. Using deception to hide things from hackers: Processes, principles, and techniques. *Journal of Information Warfare*, 5(3):26–40, 2006. *(pp. 78 and 162)*

Zhang, Ming, Boyi Xu, and Dongxia Wang. An anomaly detection model for network intrusions using one-class SVM and scaling strategy. In *Proceedings of the 11th EAI International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 267–278. Springer, 2015. *(p. 102)*

Zhang, Zheng, Jun Li, CN Manikopoulos, Jay Jorgenson, and Jose Ucles. HIDE: A hierarchical network intrusion detection system using statistical preprocessing and neural network classification. In *Proceedings of the 2nd IEEE Workshop on Information Assurance and Security*, pages 85–90, 2001. *(p. 80)*

Zheng, Wei, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. JustRunIt: Experiment-based management of virtualized data centers. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, article 18, 2009. *(p. 153)* *(p. 147)* Zhou, Lina, Judee K. Burgoon, Douglas P. Twitchell, Tiantian Qin, and Jay F. Nunamaker Jr. A comparison of classification methods for predicting deception in computer-mediated communication. *Journal of Management Information Systems (JMIS)*, 20(4): 139–166, 2004.

(pp. 43, 153, 154, and 155) Zhu, David Yu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review (OSR)*, 45(1):142–154, 2011.

VITA

Frederico Araujo was born on November 22, 1982, in Campinas, São Paulo, Brazil, where he spent his childhood playing games with his siblings and cousins, listening attentively to his grandfather's stories, and cooking with his grandmother—mostly century-old family recipes. Always curious about how things work, Frederico loved to disassemble and reassemble things, from toys to auto parts. This curiosity led him to pursue a technical high-school degree at the University of Campinas, where he was first exposed to programming languages, which he used to automate computer-aided design tasks and operate numerical control machines.

In 2002, Frederico was accepted to the University of São Paulo to pursue an Electrical and Computer Engineering degree. In the following year, he received a scholarship to investigate the use of artificial neural networks to predict component failures in distributed systems. As his passion for discovery and software engineering continued to flourish, his academic achievements granted him a full scholarship to pursue a concurrent master's degree at the Ecole Centrale Paris in France, where he consolidated his engineering background, and fell in love with the French language and culture. Two years later, Frederico completed his engineering degree at the University of São Paulo and won the Best Senior Thesis Award for excellence in undergraduate research. Upon graduation, he joined Siemens as a software engineer, where he designed and developed software frameworks, and led large projects on manufacturing and warehouse management and control systems.

It was, then, in 2010, that Frederico married his longtime sweetheart and love of his life, Rubia, and matriculated to The University of Texas for the continuation of his graduate education. There, after earning a Master of Science degree in

Computer Science, he was introduced by professor Kevin Hamlen to the exciting field of language-based security, which led him to apply his design skills to conceptualize and explore a new and fascinating science of language-based software cyber deception. After an internship at IBM Research in 2015, he completed his Doctor of Philosophy degree by designing and implementing a suite of language-based technologies that equip server software with deceptive attack-response and disinformation capabilities.

In the Fall of 2016, Frederico will continue his research career by beginning work as a member of the research staff at the IBM T.J. Watson Research Center in Yorktown Heights, NY.