INFERENCE OF RESIDUAL ATTACK SURFACE UNDER MITIGATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Kyriakos K. Ispoglou

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

Dr. Mathias Payer, Chair

Department of Computer Science

Dr. Byoungyoung Lee

Department of Computer Science

Dr. Samual Wagstaff

Department of Computer Science

Dr. Benjamin Delaware

Department of Computer Science

Approved by:

Dr. Voicu S. Popescu

Head of the Department Graduate Program

To my dad, Konstantinos.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Dr. Mathias Payer, for his astonishing work, his invaluable guidance and the –not so easy– task of advising me. I would also like to thank my co-authors Trent Jaeger, Bader AlBassam, Daniel Austin, and Vishwath Mohan for helping me with my research projects. I have to admit that this PhD would not be done without the continuous support and motivation from my family; my parents Konstantinos and Parthena and my siblings Alexandra and George. Last but not least, I would like to thank my two wonderful friends Eugenia Kontopoulou and Marios Papamichalis for the nice memories that I had with them in West Lafayette. I will be forever grateful.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

ABBREVIATIONS

ABI	Application Binary Interface
A^2DG	Abstract API Dependence Graph
API	Application Programming Interface
ARP	Arbitrary Read Primitive
ASLR	Address Space Layout Randomization
AWP	Arbitrary Write Primitive
AV	Anti-Virus
BID	Block IDentifier
BOP	Block Oriented Programming
BOPC	Block Oriented Programming Compiler
CFG	Control-Flow Graph
CFI	Control-Flow Integrity
CPU	Central Processing Unit
CVE	Common Vulnerability and Exposure
DEP	Data Execution Prevention
DFI	Data-Flow Integrity
IDS	Intrusion Detection System
IPC	Inter Process Communication
IPS	Intrusion Prevention System
IR	Intermediate Representation
JOP	Jump Oriented Programming
LLVM	Low Level Virtual Machine
ROP	Return Oriented Programming
SPL	SPloit Language

ABSTRACT

Ispoglou, Kyriakos K. PhD, Purdue University, May 2019. Inference of Residual Attack Surface under mitigations. Major Professor: Mathias Payer.

Despite the broad diversity of attacks and the many different ways an adversary can exploit a system, each attack can be divided into different phases. These phases include the *discovery* of a vulnerability in the system, its *exploitation* and the achieving *persistence* on the compromised system for (potential) further compromise and future access. Determining the exploitability of a system –and hence the success of an attack– remains a challenging, manual task. Not only because the problem cannot be formally defined but also because advanced protections and mitigations further complicate the analysis and hence, raise the bar for any successful attack. Nevertheless, it is still possible for an attacker to circumvent all of the existing defenses –under certain circumstances.

In this dissertation, we define and infer the *Residual Attack Surface* on a system. That is, we expose the limitations of the state-of-the-art mitigations, by showing practical ways to circumvent them. This work is divided into four parts. It assumes an attack with three phases and proposes new techniques to infer the Residual Attack Surface on each stage.

For the first part, we focus on the vulnerability discovery. We propose *FuzzGen*, a tool for automatically generating fuzzer stubs for libraries. The synthesized fuzzers are target specific, thus resulting in high code coverage. This enables developers to expose and fix vulnerabilities (that reside deep in the code and require initializing a complex state to trigger them), before they can be exploited. We then move to the vulnerability exploitation part and we present a novel technique called *Block Oriented Programming* (BOP), that automates data-only attacks. Data-only attacks defeat advanced control-flow hijacking defenses such as Control Flow Integrity. Our framework, called *BOPC*, maps arbitrary exploit payloads into execution traces and encodes them as a set of memory writes. Therefore an attacker's

intended execution “sticks” to the execution flow of the underlying binary and never departs from it. In the third part of the dissertation, we present an extension of BOPC that presents some measurements that give strong indications of what types of exploit payloads are *not* possible to execute. Therefore, BOPC enables developers to test what data an attacker would compromise and enables evaluation of the Residual Attack Surface to assess an application’s risk. Finally, for the last part, which is to achieve persistence on the compromised system, we present a new technique to construct arbitrary malware that evades current dynamic and behavioral analysis. The desired malware is split into hundreds (or thousands) of little pieces and each piece is injected into a different process. A special emulator coordinates and synchronizes the execution of all individual pieces, thus achieving a “distributed execution” under multiple address spaces. malWASH highlights weaknesses of current dynamic and behavioral analysis schemes and argues for full-system provenance.

Our envision is to expose all the weaknesses of the deployed mitigations, protections and defenses through the Residual Attack Surface. That way, we can help the research community to reinforce the existing defenses, or come up with new, more effective ones.

1 INTRODUCTION

As computer systems evolve, they become a prominent target for all kinds of attacks. Some attacks such as control-flow hijacking, can have devastating effects as they force the system to execute arbitrary, attacker chosen, code. Hence, a plethora of defense mechanisms have been proposed to mitigate the impact, or even prevent those attacks. This started an arms race, with the attackers finding new ways to circumvent the existing protections and mitigations, and defenders reinforcing their defenses and creating new ones [1]. However, as defenses evolve, attacks become sophisticated, more subtle and harder to successfully execute.

Despite the stunning success of the applied defenses, they all have some form of limitation or weakness that in some cases, an adversary can leverage. This is sufficient to leave some small space for a successful attack. Nevertheless, most attacks remain infeasible, thus significantly raising the bar against successful compromise.

Therefore, given all these protections and mitigations applied on a system, what are the *remaining* options for an attacker? This dissertation defines this as the *Residual Attack Surface* i.e., the attack possibilities for an adversary on a system where a set of mitigations and protections are applied.

This dissertation focuses on defining and inferring the *Residual Attack Surface*. Ideally, to precisely measure the Residual Attack Surface, one has to identify all possible vulnerabilities in the system, determine which ones are exploitable, and try to execute each exploitable vulnerability to get unauthorized access to the system. Finally, if the attacker does not have the desired privileges on the system, or she wants to utilize the system as a pivot to attack other systems, she needs to repeat these steps for each target. Clearly, each of the aforementioned steps requires an enumeration of all potential inputs to the system, which are uncountable. Furthermore the problem of determining whether a vulnerability is exploitable or not is undecidable (see Appendix 7.1).

Hence, to infer the Residual Attack Surface, we use the following approach: *First*, we show that the Residual Attack Surface *does exist* and it is nonzero. We find vulnerability locations and we present new techniques that circumvent all of the deployed state-of-the-art mitigations (either combined or alone). *Second*, we find "upper bounds" on attacker's *capabilities*. That is, we aim to identify the *limits* of exploitation, i.e., what an adversary is capable of doing in the best case scenario for her (which is also the worst case scenario for the system).

However, there are some concerns with our approach. Due to the large number of potential attacks, the inference of the Residual Attack Surface becomes complicated. For instance, there can be uncountable ways to break into a system, as each system is different. This implies that creating general techniques is challenging, as each system can have its own Residual Attack Surface.

To deal with this challenge, we *divide* an attack into distinct phases and we infer the Residual Attack Surface in each phase. Despite the large diversity of the potential attacks, these phases are common in most attacks. Furthermore, dividing an attack into distinct phases has several advantages from a defender's point of view. *First*, defenses become more targeted (and therefore more effective) as they aim to protect only one part of the system. *Second*, multiple defenses, orthogonal to each other, can be applied together at different phases. This multi-layered approach implies that the adversary has to defeat several layers of protection before she breaks into the system, so any attack becomes significantly more complicated. Although there can be multiple ways to divide an attack into phases, this dissertation assumes a three phase, coarse grained separation.

1.1 The three phases of an attack

As stated above, this dissertation divides an attack into three distinct phases as shown in Figure 1.1. We do not state that this is the only way to divide an attack; more fine-grained divisions may exist. However, the seperation suits the needs of inference of the Residual Attack Surface. These three phases are:

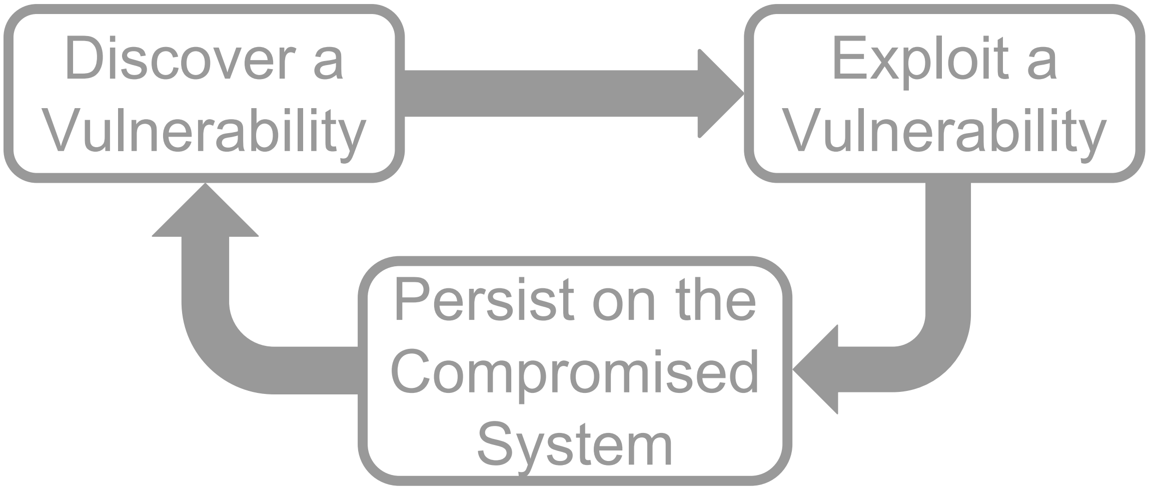


Figure 1.1.: The three phases of an attack.

- The *discovery* of a vulnerability
- The *exploitation* of a vulnerability
- The *persistence* on the compromised system

An attack starts with *reconnaissance*. That is, the adversary first gets some knowledge on the system, how its components operate, what their weaknesses are and what the potential points to attack are. The goal of this step, is for attacker to find a way to get the system into a state that it is not designed to be in. That is, to find out “what could go wrong” in the system. This is called a *bug* (or *flaw*).

In the *second* phase, the attacker is looking for a way to leverage this bug in order to compromise the system. That is, taking advantage of the bug, to force the system to perform actions on the attacker’s behalf (e.g., execute the attacker’s indented code). This is commonly referred to as *exploitation*.

Although in some scenarios it is sufficient to – successfully – exploit a bug, this may not be enough to complete an attack, so a *third* phase is required. For instance, the attacker may be able to execute her own code, but with low privileges. Thus, the attacker needs to establish herself on the compromised system and repeat the same process. This “persistence” on the compromised system is a crucial, as it allows the attacker to continue attacking further.

This is done by either using the compromised system as a pivot to reach other systems that are not otherwise reachable, or to further compromise another component on the same system and elevate her privileges.

1.1.1 Discovering a vulnerability

In the first phase of the attack the adversary analyzes the target system and tries to trigger "abnormal" behavior, which indicates that the system has reached an *undefined* state. An undefined state, is a state that the system is not supposed or designed to be in. This usually happens, when the system accepts unexpected input it cannot handle properly. Then the attacker can abuse this behavior to start manipulating the system state.

For instance, consider a system that plays *blackjack*. A player gives a bet as input to the system and the game starts. When the player wins, the bet is added to her balance. When the player loses, the bet is subtracted from her balance. Now consider a situation where the player gives a *negative* bet and loses intentionally. Let us assume that the designers of the system did not consider this case –as it is not feasible in reality to bet a negative amount of money– and hence, they did not sanitize the input properly. However the system continues to operate normally, so when the player loses, the system subtracts the (negative) bet from player's balance. The result is that player's balance is increased as the subtraction of a negative number results in an addition. Hence, an evil player can abuse this system to always win.

Technically, this behavior is referred as crash, bug, flaw, or vulnerability. Although each of these terms indicates that the system has reached an "undefined" state, they all describe different situations.

A *crash* indicates that the system has stopped working. The root cause for it is usually an *access violation* or a *segmentation fault*. For instance when the system tries to do some operations on a pointer that does not point to valid memory address, the underlying operating system, throws an exception which results in program termination.

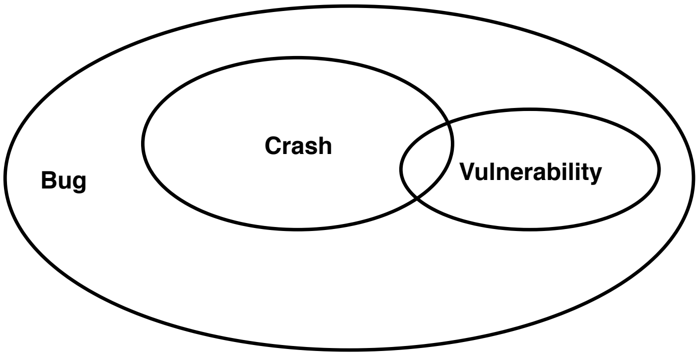


Figure 1.2.: A visual representation of a crash, a bug and a vulnerability.

A *bug*, may cause a crash. Segmentation faults and memory errors are types of bugs that can crash the system. However, a bug is also anything that results in an *undefined behavior* in the system. For example, if an attacker can manipulate a timeout parameter on a system, she can cause the system to hang, by supplying a very large value. Although the program still continues to work, it stops responding, which in practice means that it is dead. Back in our blackjack example, the system has a bug that does not cause any crashes. A player can bet negative amounts of money while the system continues to operate normally.

A *flaw* is a special type of bug that is created during sysstem design and carried into its implementation. It is usually referred as *logic error*. For instance, a system that implements a weak access control policy and allows low privileged users to access functionality from super users has a design flaw. Although the system does not crash, the results can be devastating.

Finally, a *vulnerability* is a bug that allows the attacker to manipulate the system. That is, a vulnerability allows unintended state changes on the system that are controlled by the attacker. Therefore, a vulnerability is a bug that the attacker can leverage to control the program's state. As an example, if a system tries to access a pointer that points to an invalid location, this is a bug. However, if an attacker can first control this pointer and make it point to a memory region that contains executable code instead of crashing, the system will start executing code that it is not supposed to execute (this type of attack is called control-flow hijacking). Figure 1.2 shows a visual representation of a crash, a bug, and a vulnerability.

From the aforementioned types of issues, an adversary is interested in finding vulnerabilities as it is the basic requirement to get unauthorized access to the system. Nevertheless, there are some scenarios that the attackers can leverage bugs to cause *Denial of Service* (DoS) attacks and compromise the availability of the system.

Finding vulnerabilities has been an interesting problem for the research community for several years. During that time many techniques have been developed for finding vulnerabilities. One of the oldest and most effective methods is *fuzzing*. Fuzzing is the process of supplying random and unexpected input to program and looking for abnormal behavior (i.e., crashes or hangs). Although fuzzing is based on a simple and “naive” concept, it turned out to be much more efficient in practice than other sophisticated approaches that involve symbolic execution [2] or SMT [3] solvers. Despite its simplicity, making fuzzing efficient is challenging. Some bugs reside deep in the code and it is hard to craft proper input to trigger them. Discovering new techniques to improve fuzzing (in both directions of performance and effectiveness) has received significant attention from the research community. Over the last years there is a lot of great work in the area, such as TFuzz [4]. Recent work of *Klees et al.* [5] compares and evaluates the 32 most recent fuzzing techniques, showing their effectiveness in bug finding.

Although attackers utilize fuzzing to find vulnerabilities, fuzzers are also used with great success as defense mechanisms. Finding and eliminating bugs is important as it protects against their potential exploitation. While an attacker only needs to find a *single* bug to exploit, defenders need to find *all* bugs before attackers. However, defenders have the advantage of knowing the source code and deployment, which is beneficial for fuzzing.

1.1.2 Exploiting a vulnerability

Given a bug on the target system, an attacker needs to find a way to turn it into a vulnerability and synthesize an exploit that gives her unauthorized access to the system. However, determining if a bug is exploitable is a challenging, manual task. Despite the great effort [6, 7] that has been done towards automating exploit generation, this problem remains unsolved. Furthermore, the wide deployment of mitigations against control-flow hijacking attacks such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), stack canaries, Control Flow Integrity (CFI), and shadow stacks, further complicate this task.

One of the oldest techniques to exploit a vulnerability is *code injection*. The attackers supply arbitrary code to the program and force it to execute this code through a vulnerability. *Data Execution Prevention* (DEP) [8] stops code injection by leveraging memory page permissions to make pages that contain contain data non-executable. An attacker is still able to inject code, but when control is transfered to it, a segmentation fault occurs.

However, injecting new code into a program is not always necessary. Most of the times, the program itself contains sufficient code that it is possible for an attacker to *reuse* it and synthesize the desired payload. This idea started with the *return-to-libc* [9] attacks and extended to its general form, called *Return Oriented Programming* (ROP) [10]. In addition to ROP, some variations have been proposed, with *Jump Oriented Programming* [11] and ROP without returns [12] being two of them.

To defeat code reuse attacks, one has to look into the execution flow of an attacker's payload. Code reuse attacks, execute small pieces of code called *gadgets*, chained together. However, this style of execution violates the benign control flow of the program. *Control Flow Integrity* (CFI) [13] prevents code reuse attacks, by sanitizing the target address of each indirect control flow transfer. CFI leverages the program's Control Flow Graph (CFG) to identify the *forward* edges and inserts additional instrumentation code to check the integrity of the runtime target address. Similarly, *Shadow Stacks* [14] assure the integrity of the *backward* edges. Even though CFI significantly raises the bar for successful exploitation, it still suffers from some weaknesses. First, CFI overapproximates the allowed target set, thus giving some freedom to an attacker [15]. Furthermore, *Data-Only* attacks [16, 17] are also feasible as they do not violate a program's CFG. However those attacks violate the program's *Data Flow Integrity* (DFI) [18]. Unfortunately, proposed DFI mechanisms have high overhead, which prevents them from being deployed.

Another orthogonal mitigation is *Address Space Layout Randomization* (ASLR) [19]. ASLR randomizes the various sections of a program when loaded in memory. Thus, the attacker does not know the exact address of the desired code to execute and hence, she cannot transfer control to it. It is possible for the attacker to guess the correct address, but the probability is negligible for realistic scenarios. The main limitation here, is that the system randomizes *whole* sections of the program. Thus, if an attacker knows one address from a section, it can compute the relative offset and find the exact address of her code [20]. Therefore, an attacker only needs to find an *information leak* (i.e., leak an address). Although there are some improvements on ASLR, such as ASLR-Guard [21] that aims to prevent information leaks, it is still remains vulnerable to information leaks.

1.1.3 Persisting on the compromised system

At this point the attacker has unauthorized access (i.e., she is able to execute her own code) to the –compromised– system. In many cases this is not what the attacker really wants, as code may run with limited privileges. Hence, the attacker needs to find a robust and reliable way to "establish" herself on the system. That is, to install some *backdoor* to ease future access. This enables the attacker to either: *i)* utilize system as a pivot to launch new attacks on other systems that are not directly accessible otherwise, *ii)* attack different components on the same system to elevate privileges, or *iii)* simply have access for any potential future use. Therefore, attacker needs to install code with *malicious* indention on the system. This code is referred as *malicious software* or *malware* for short.

Detecting malware and inferring whether a program performs malicious actions or not, is an open problem with a lot and interesting ongoing research. Recent work on malware detection [22, 23] makes the task of evading detection challenging. Detection is based on two main approaches: static and dynamic. Special monitoring programs such as AnitVirus (AV), Intrusion Detection Systems (IDS) or Intrusion Prevention Systems (IPS), run – with high privileges – on the system and inspect it for any suspicious activity. However, AV software nowadays is reinforced with IDS and IPS capabilities, so there is not clear distinction on them. *AntiVirus* software focuses on detecting malicious files. It periodically scans the filesystem and inspects executable files that are about to run. This is done either by statically analyzing the file, or using *emulators* to run the program in a virtual environment and carefully monitor its activity. *Intrusion Detection/Prevention Systems* focus on monitoring and inspecting the *behavior* of the applications running on the system.

Static detection methods [23, 24, 25] analyze programs without executing them. One of the oldest –and most successful– techniques is signature detection. In *signature detection*, AV extracts “patterns” and computes special “signatures” [1] from the target program, which uses them to lookup in a huge database of all known malware. Furthermore, AV performs a sequence of various analyses on the file looking for notorious system calls, self-modifying or obfuscated code, and so on. However, attackers can defeat static detection through *metamorphic* [26] malware. Metamorphic malware modifies itself each time it gets propagated. That is, the same malware can have an infinity amount of different instances thus thwarting signature detection.

Dynamic detection methods [22, 27, 28, 29, 30, 31, 32, 33, 34, 35] on the other hand, focus on malware’s behavior instead. They let the malware run until it reveals its real indentions. The main limitation of static detection is that it is easy for an attacker to thwart analysis by obfuscating the code, applying anti-disassembly [36] tricks or creating metamorphic instances. The intuition behind this concept is that even though two instances of a metamorphic malware are very different they still have the exact same behavior. Nevertheless there are some techniques to evade dynamic detection [37, 38, 39] which keeps malware detection an interesting, open research problem.

1.2 Dissertation Statement

This dissertation infers the Residual Attack Surface at each of the three phases of an attack. It presents state-of-the-art techniques that a defender can utilize in her analysis to defend. Although it is hard to precisely measure the Residual Attack Surface, its objective is

[1] A signature comes in the form of a *hash*.

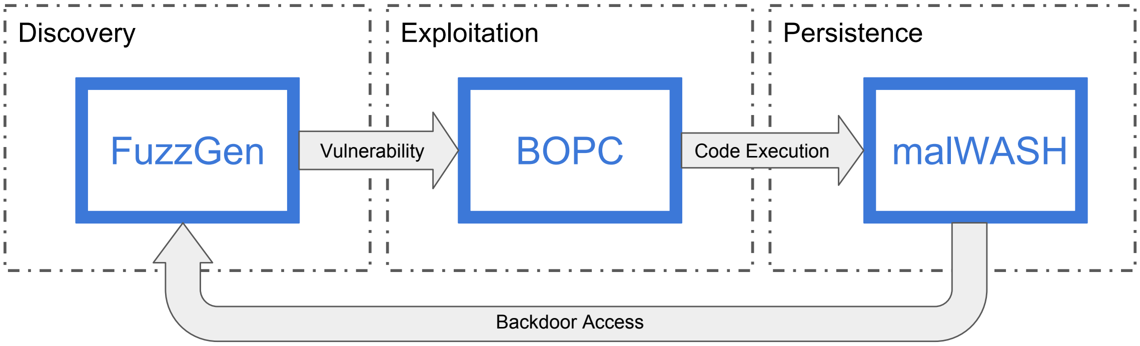


Figure 1.3.: The interconnection of dissertation's three main components.

to expose weaknesses in existing mitigations and assist the research community to improve existing, or come up with new, stronger defenses. The dissertation statement is shown below:

> *The wide and successful deployment of mitigations led to the development of highly sophisticated attacks that are challenging to orchestrate. The Residual Attack Surface consists of the set of attacks that remain feasible and practical. It provides strong indications on what an adversary is capable of, thus assisting defenders in exposing the limitations of existing protections.*

The dissertation consists of three major components. Each part demonstrates a new, practical technique that an adversary can utilize in each phase of an attack as described in Section 1.1. The interconnection of these components is shown in Figure 1.3.

The first part of the dissertation focuses on finding memory corruption vulnerabilities, by using target-specific fuzzing. This is done through an automatic fuzzer generation framework, called *FuzzGen* [40]. FuzzGen, leverages the source code of a given application library, to automatically generate a fuzzer for it. The resulting fuzzers achieve deep code coverage, thus exposing bugs that are otherwise, hard to reach. Furthermore, large scale fuzzing becomes simple, as we automatically create specific fuzzers for every application that we want to fuzz. FuzzGen assists defenders to quickly find bugs that are potentially exploitable, and prevent attackers from exploiting them.

In the second part (vulnerability exploitation), we propose a novel technique called *Block Oriented Programming* (BOP) [17] which automates data-only attacks. BOP, leverages a memory corruption vulnerability (found in the previous step), to automatically generate arbitrary and Turing-complete, data-only exploit payloads. BOP comes with a framework, called BOPC (*BOP Compiler*) that demonstrates proof-of-concept exploits for several vulnerable applications, protected through state-of-the-art control flow hijacking defenses such as CFI and shadow stacks. BOP can help software developers to highlight payloads that an attacker is still capable of executing under a heavily protected (and constrained) environment. For instance, defenders can test whether a bug at a particular statement enables a practical code reuse attack in the program.

The last part, focuses on achieving persistence in the compromised system without triggering any alarms or suspicious behavior. To evade detection and show that Residual Attack Surface does exists, a new technique is proposed capable of constructing arbitrary malware that evades all kinds of dynamic and behavioral analysis. Attacker's payload is "chopped" into hundreds of little pieces with each piece injected into the address space of a different process. A special process, called *emulator*, coordinates and synchronizes the execution of all individual pieces, thus achieving a "distributed execution" under multiple address spaces. Our framework, called *malWASH* [37] automates this process. malWASH, reveals a new direction on stealthy malware. So far, malware detection mechanisms do not consider a *distributed* malware execution and operate on a *single* process. Malware analysts can utilize malWASH framework to evaluate and reinforce their detection mechanisms so they can detect that kind of dangerous malware.

Finally, this dissertation closes with a discussion of future work. There are many unexplored dimensions on the area of Residual Attack Surface while this dissertation only shines some light on it. The aim of this dissertation is to assist the research community to make existing defense mechanisms stronger and come up with new ones, by exposing the weaknesses of the existing mitigations and protections.

1.3 Dissertation Organization

This dissertation presents the work on inferring the Residual Attack Surface. The overall organization is shown below:

- Chapter 1 has introduced the attack phases, the intuition behind the Residual Attack Surface and why it is useful as long the main approaches to infer it.

- Chapter 2 describes *FuzzGen*, a new technique to find vulnerabilities in library code through automatic fuzzer synthesis.

- Chapter 3 provides a detailed explanation behind the concept of *Block Oriented Programming*, and provides the design, implementation and evaluation of *BOPC*, a framework built on BOP that automates data-only attacks.

- Chapter 4 is an ongoing work, that extends BOPC to assess exploitation capabilities. Our tool, *X-Cap* provides strong indications on the exploit payloads that an attacker can execute on a vulnerable application through Block Oriented Programming.

- Chapter 5 presents a new technique to evade dynamic and behavioral analysis, through distributed malware execution.

- Chapter 6 discusses the related research –which is the motivation to this work– as well as some of the new directions that this research can continue.

- Chapter 7 concludes the dissertation.

2 FUZZGEN: AUTOMATIC FUZZER GENERATION

Fuzzing is a testing technique to discover unknown vulnerabilities in software. When applying fuzzing to libraries, the core idea of supplying random input remains unchanged, yet it is non-trivial to achieve good code coverage. Libraries cannot run as standalone programs, but instead are invoked through another application. Triggering code deep in a library remains challenging as specific sequences of API calls are required to build up the necessary state. Libraries are diverse and have unique interfaces that require unique fuzzers, so far written by a human analyst.

To address this issue, we present FuzzGen, a tool for automatically synthesizing fuzzers for complex libraries in a given environment. FuzzGen leverages a *whole system analysis* to infer the library's interface and synthesizes fuzzers specifically for that library. FuzzGen requires no human interaction and can be applied to a wide range of libraries. Furthermore, the generated fuzzers leverage LibFuzzer to achieve better code coverage and expose bugs that reside deep in the library.

FuzzGen was evaluated on Debian and the Android Open Source Project (AOSP) selecting 7 libraries to generate fuzzers. So far, we have found 17 previously unpatched vulnerabilities with 6 assigned CVEs. The generated fuzzers achieve an average of 54.94% code coverage; an improvement of 6.94% when compared to manually written fuzzers, demonstrating the effectiveness and generality of FuzzGen.

2.1 Introduction

Modern software distributions like Debian, Ubuntu, and the Android Open Source Project (AOSP) are large and complex ecosystems with many different software components. Debian consists of a base system with hundreds of libraries, system services and their configurations, and a customized Linux kernel. Similarly, AOSP consists of the ART virtual machine, Google's support libraries, and several hundred third party components including open source libraries and vendor specific code. While Google has been increasing efforts to fuzz test this code, e.g., OSS-Fuzz [41, 42], code in these repositories does not always go through a rigorous code review process. All these components in AOSP may contain vulnerabilities and could jeopardize the security of Android systems. Given the vast amount of code and its high complexity, fuzzing is a simple yet effective way of uncovering unknown vulnerabilities [43, 44]. Discovering and fixing new vulnerabilities is a crucial factor in improving the overall security and reliability of Android.

Automated generational grey-box fuzzing, e.g., based on AFL [45] or any of the more recent advances over AFL such as AFLfast [46], AFLGo [47], collAFL [48], Driller [49], VUzzer [50], T-Fuzz [4], QSYM [51], or Angora [52] are highly effective at finding bugs in programs by mutating inputs based on execution feedback and new code coverage [5]. Programs implicitly generate legal complex program state as fuzzed input covers different program paths. Illegal paths quickly result in an error state that is either gracefully handled by the program or results in a true crash. Code coverage is therefore an efficient indication of fuzzed program state.

While such greybox-fuzzing techniques achieve great results regarding code coverage and number of discovered crashes in *programs*, their effectiveness does not transfer to fuzzing *libraries*. Libraries expose an API without dependency information between individual functions. Functions must be called in the right sequence with the right arguments to build complex state that is shared between calls. These implicit dependencies between library calls are often mentioned in documentation but are generally not formally specified. Calling random exported functions with random arguments is unlikely to result in an efficient fuzzing campaign. For example, *libmpeg2* requires an allocated context that contains the current encoder/decoder configuration and buffer information. This context is passed to each subsequent library function. Random fuzzing input is unlikely to create this context and correctly pass it to later functions. Quite the contrary, it will generate a large number of false positive crashes when library dependencies are not enforced, e.g., the configuration function may set the length of the allocated decode buffer in the internal state that is passed to the next function. A fuzzer that is unaware of this length field may supply a random length, resulting in a spurious buffer overflow. Alternatively, “invalid state checks” in library functions will likely detect dependency violations and terminate execution early, resulting in wasted fuzzing performance. To effectively fuzz libraries, a common approach is to manually write small programs which build up state and call API functions in a “valid” sequence. This allows the fuzzer to build up the necessary state to test functionality deep in the library.

libFuzzer [53] facilitates library fuzzing through the help of an analyst. The analyst writes a small “fuzzer stub”, a function that (i) calls the required library functions to set up the necessary state and (ii) leverages random input to fuzz state and control-flow. The analyst must write such a stub for each tested component. Determining interesting API calls, API dependencies, and fuzzed arguments is at the sole discretion of the analyst. While this approach mitigates the challenge of exposing the API, it relies on *deep* human knowledge of the underlying API and its usage. Hence, this approach does not scale to many different libraries.

FuzzGen is based on the following intuition: *existing code* on the system *utilizes* the library in *diverse* aspects. Abstracting the graph of possible library dependencies allows us to infer the complex library API. Different aspects of the API are tested by *automatically generating custom fuzzer stubs* based on the inferred API. The automatically generated fuzzers will execute sequences of library calls that are similar to those present in real programs without the “bloat” of real programs, i.e., removing all computation that is not strictly necessary to build the state required for fuzzing. These fuzzers will achieve deep coverage, improving over fuzzers written by an analyst as they consider real deployments and API usage.

On one hand, many libraries contain unit tests that exercise simple aspects of the library. On the other hand, programs that utilize a library’s API build up deep state for specific functions. Leveraging only individual test cases for fuzzing is often too simplistic and building on complex programs results in low coverage as all the program functionality is executed alongside the target library. Test cases are too simple and fail to expose deep

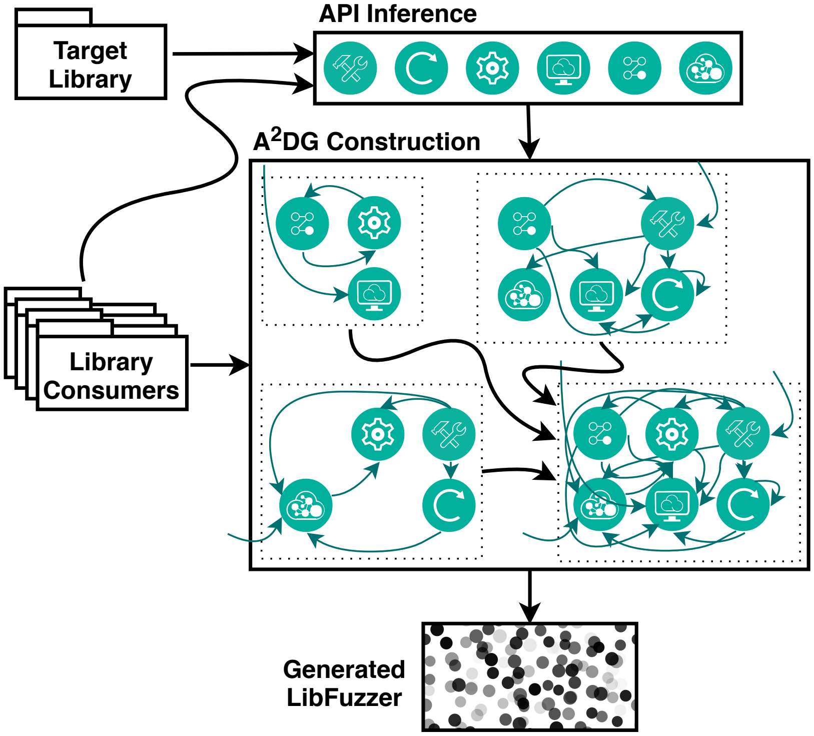


Figure 2.1.: The main intuition behind FuzzGen. To synthesize a fuzzer, FuzzGen performs a whole system analysis to extract all valid API interactions.

bugs while full programs are too complex. A mechanism that automatically constructs arbitrarily complex fuzzer stubs with complex API interactions and library state allows sufficient testing of complex API functions. The set of all test cases and programs which use a library covers nearly all relevant API invocations and contains code to set up the necessary complex state to execute API calls. The vast amount of different library usages implicitly defines an *Abstract API Dependence Graph* (A^2DG). Based on this A^2DG it is possible to automatically create fuzzer stubs that test different aspects of a library (Figure 5.1).

To address the challenges of fuzzing complex libraries, we propose *FuzzGen*. FuzzGen consists of three parts: an API inference, an A^2DG construction mechanism, and a fuzzer generator that leverages the A^2DG to produce a custom libFuzzer "fuzzer stub". The API inference component builds an A^2DG based on all test cases and programs on a system that use a given library. The A^2DG is a graph that records all API interactions, including parameter value range and possible interactions. Our analysis infers library use and constructs a generic A^2DG based on this use. The fuzzer generator synthesizes fuzzers that build up complex state and leverage fuzz input to trigger faults deep in the library. FuzzGen automates the manual process of the analyst in creating custom-tailored fuzzers for libraries and specific library functions. The key contribution of FuzzGen is an *automatic* way to create new libFuzzer [53] stubs, enabling *broad and deep* library fuzzing.

FuzzGen performs a *whole system analysis*, iterating over all programs and libraries that use the target library to infer the A^2DG. It then *automatically* generates fuzzer stubs (ranging from $1,000$ to $10,000$ LoC) that encode the A^2DG and use libFuzzer to fuzz individual API components. FuzzGen was evaluated on Debian and Android [54].

Our evaluation of FuzzGen so far, resulted in 17 discovered vulnerabilities in the Android media framework, with 6 assigned CVEs: CVE-2019-2176 [55], CVE-2019-2108 [56], CVE-2019-2107 [57] and CVE-2019-2106 [58] (critical), CVE-2017-13187 [59] (high) and –duplicate– CVE-2017-0858 [60] (medium). (in Section 2.8 we provide more details on these vulnerabilities). Finding and eliminating vulnerabilities in these components is crucial to prevent potential attacks such as StageFright [61]. So far, FuzzGen has reported 17 new vulnerabilities in Android native libraries and Debian. The discovered bugs range from denial of service to stack buffer overflows, as shown in Section 3.6. Overall FuzzGen makes the following contributions:

- Design of a *whole system analysis* that infers valid API interactions for a given library based on existing programs and libraries that use the target library—abstracting the information into an Abstract API Dependence Graph (A^2DG);
- Based on the A^2DG, FuzzGen creates libFuzzer stubs that construct complex program state to expose vulnerabilities in deep library functions was developed—fuzzers are generated without human interaction;
- Evaluation of the prototype on AOSP and Debian demonstrates the effectiveness and the generality of the FuzzGen technique. Generating fuzzers for 7 libraries, FuzzGen discovered 17 bugs. The generated fuzzers achieve 54.94% code coverage on average, compared to 48.00% that fuzzer stubs—written manually by experts—achieve.

A note on disclosure: All bugs have been responsibly disclosed, and fixes have been pushed to the corresponding projects. The source code of our prototype is available at `https://github.com/HexHive/FuzzGen`, allowing other researchers to reproduce our results and to extend our automatic fuzzer generation technique.

2.2 The case for API-aware fuzzer construction

Writing an effective API-aware fuzzer requires an in-depth understanding of the target library and pinpointing the interesting components for fuzzing. Consider the libmpeg2 library, which provides encoding and decoding functions for MPEG2 video streams. The library contains several functions to build up a per-stream context that other functions take as a parameter. This approach of encapsulating state is common in libraries. Figure 2.2 shows a code snippet for properly initializing an MPEG2 decoding object. A fully initialized decoder object is required to decode a video frame. Without this decoder object, frames cannot be decoded.

While a target-agnostic fuzzer (invoking all functions with random arguments in a random order) may find simple issues, deep bugs will likely be missed due to their dependence on complex state. Naive fuzzers are also prone to false positives due to lack of API awareness. Consider a fuzzer that targets frame decoding. If the context does not contain a valid length with a pointer to an allocated decode buffer then the fuzzer will trigger a false positive crash when the decoded frame is written to unallocated memory. However, this is not a bug in the decode function. It is simply improper initialization. Orthogonally, by supplying random values to certain arguments, such as function pointers or sizes, a fuzzer may trigger memory errors. These crashes do not correspond to actual bugs or vulnerabilities as such an illegal context cannot be generated through any possible execution of a benign use of the library. Inferring API dependencies, such as generating a common context, initializing the necessary buffers, and preparing it for usage, is challenging because dependencies are not encoded as part of the library specification.

```
/* 1. Obtain available number of memory records */
iv_num_mem_rec_ip_t num_mr_ip = { ... };
iv_num_mem_rec_op_t num_mr_op = { ... };
impeg2d_api_function(NULL, &num_mr_ip, &num_mr_op);

/* 2. Allocate memory & fill memory records */
nmemrecs = num_mr_op.u4_num_mem_rec;
memrec    = malloc(nmemrecs * sizeof(iv_mem_rec_t));

for (i=0; i<nmemrecs; ++i)
    memrec[i].u4_size = sizeof(iv_mem_rec_t);

impeg2d_fill_mem_rec_ip_t fill_mr_ip = { ... };
impeg2d_fill_mem_rec_op_t fill_mr_op = { ... };
impeg2d_api_function(NULL, &fill_mr_ip, &fill_mr_op);

nmemrecs = fill_mr_op.s_ivd_fill_mem_rec_op_t
                     .u4_num_mem_rec_filled;

for (i=0; i<nmemrecs; ++i)
    memrec[i].pv_base = memalign(memrec[i].u4_mem_alignment,
  memrec[i].u4_mem_size);

/* 3. Initalize decoder object */
iv_obj_t *iv_obj = memrec[0].pv_base;
iv_obj->pv_fxns  = impeg2d_api_function;
iv_obj->u4_size  = sizeof(iv_obj_t);

impeg2d_init_ip_t init_ip = { ... };
impeg2d_init_op_t init_op = { ... };
impeg2d_api_function(iv_obj, &init_ip, &init_op);

/* 4. Decoder is ready to decode headers/frames */
```

Figure 2.2.: Source code that initializes an MPEG2 decoder object. Low level details such as struct field initializations, variable declarations, or casts are omitted for brevity.

However, by observing a module that utilizes libmpeg2 (i.e., a *library consumer*), we could observe the dependencies between the API calls and infer the correct order of context initialization calls. Such dependencies come in the form of (a) *control flow* dependencies and (b) *shared* arguments (variables that are passed as arguments in more than one API call). Furthermore, arguments that hold the state of the library (e.g., the context), should not be fuzzed, but instead they should be passed, without intermediate modification, from one call to the next. Note that this type of information is usually *not formally specified*. The libmpeg2 library exposes a single API call, `impeg2d_api_function`, that dispatches to a large set of internal API functions. Yet, this state machine of API dependencies is not made explicit in the code.

2.3 Background and Related Work

Early fuzzers focused on generating random parameters to test resilience of code against illegal inputs. Different forms of fuzzers exist depending on how they generate input, handle crashes, or process information. *Generational* fuzzers, e.g., PROTOS [62], SPIKE [63], or PEACH [64], generate inputs based on a format specification, while *mutational* fuzzers, e.g., AFL [45], honggfuzz [65], or zzuf [66], synthesize inputs through random mutations on existing inputs, according to some criterion (e.g., code coverage). Typically, increasing code coverage and number of unique crashes is correlated with fuzzer effectiveness.

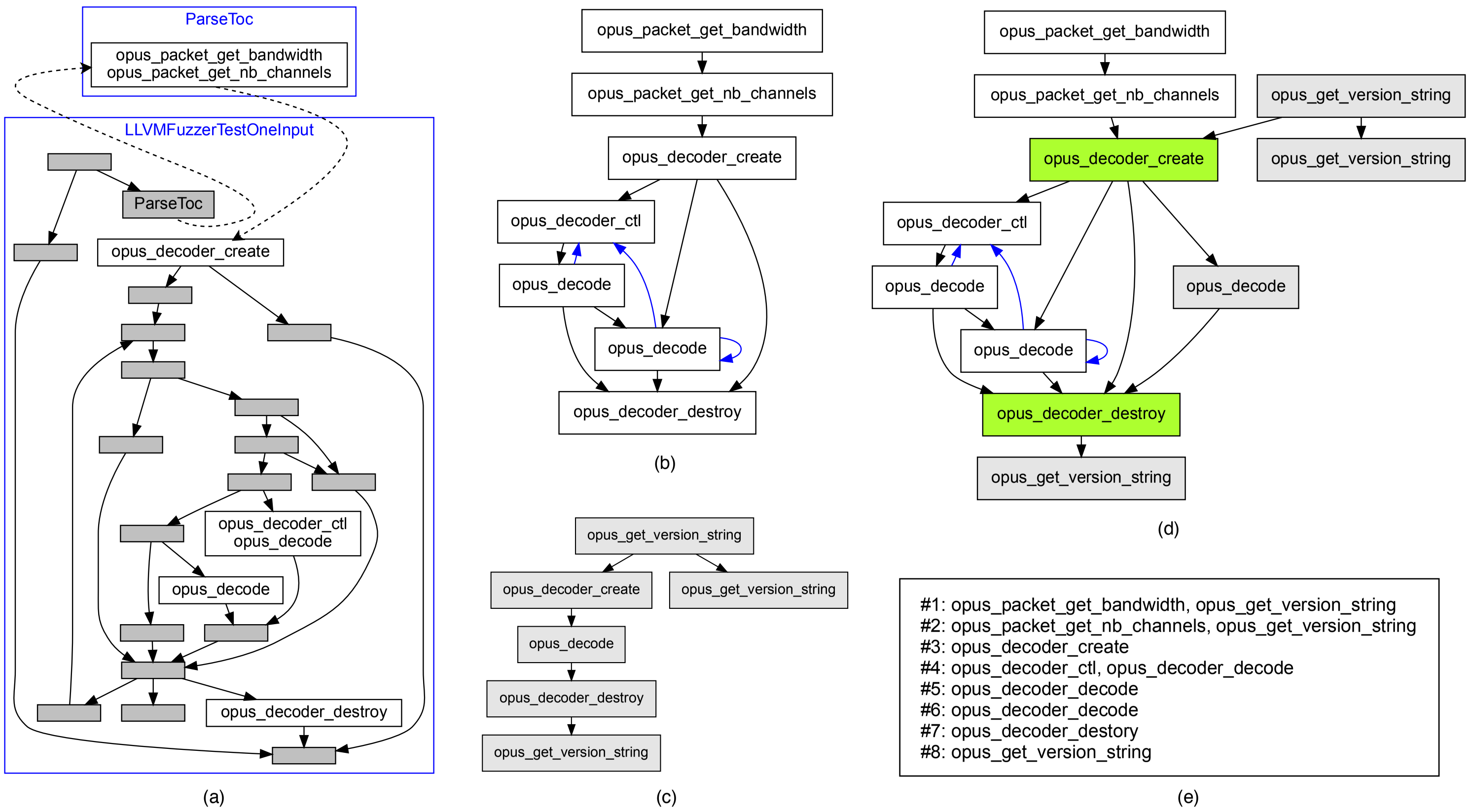
Mutational fuzzers have become the de-facto standard for fuzzing due to their efficiency and ability to adapt input. The research community developed additional metrics to classify fuzzers, based on their "knowledge" about the target program. *Blackbox* fuzzers, have no information about the program under test. That is, they treat all programs *equally*, which allows them to target arbitrary applications. *Whitebox* fuzzers are aware of the program that they test and are target-specific. They adjust inputs based on some information about the target program, targeting more "interesting" parts of the program. Although whitebox fuzzers are often more effective in finding bugs (as they focus on a small part of the program) and therefore have lower complexity, they require manual effort and analysis and allow only limited reuse across different programs (the whitebox fuzzer for program A cannot be used for program B). *Greybox* fuzzers attempt to find a balance between blackbox and whitebox fuzzing by inferring information about the program and feeding that information back to guide the fuzzing process. Evaluating fuzzers is challenging. We follow proposed guidelines [5] for a thorough evaluation.

Code coverage is often used in greybox fuzzers to determine if an input should be further evaluated. The intuition is that the more code a given input can reach the more likely is to expose bugs that reside deep in the code. Fuzzers are limited by the *coverage wall*. This occurs when the fuzzer stops making progress, and could be due to limitations of the model, input generation, or other constraints. Any newly generated input will only cover code that has already been tested. Several recent extensions of AFL have tried to address the coverage wall using symbolic or concolic execution techniques [2] and constraint solving. Driller [49] detects if the fuzzer no longer increases coverage and leverages program tracing to collect constraints along paths. Driller then uses a constraint solver to construct inputs that trigger new code paths. Driller works well on CGC binaries but the constraint solving cost can become high for larger programs. VUzzer [50] leverages static and dynamic analysis to infer control-flow of the application under test, allowing it to generate application-aware input. T-Fuzz [4] follows a similar idea but instead of adding constraint solving to the input generation loop, it rewrites the binary to bypass hard checks. If a crash is found in the rewritten binary, constraint solving is used to see if a crash along the same path can be triggered in the original binary. FairFuzz [67] increases code coverage by prioritizing inputs that reach "rare" (i.e., triggered by very few inputs) areas of the program, preventing mutations on checksums or strict header formats. FuzzGen addresses the coverage wall by generating multiple different fuzzers with different API interactions. The A^2DG allows FuzzGen to quickly generate alternate fuzz drivers that explore other parts of the library under test.

Although the aforementioned fuzzing approaches are effective in exposing unknown vulnerabilities, they assume that the target program has a well defined interface to supply random input and observe for crashes. These methods cannot be extended to deal with libraries. A major challenge is the *interface diversity* of the libraries, where each library provides a different interface through its own set of exported API calls. DIFUZE [68] was the first approach for interface-aware fuzzing of kernel drivers. Kernel drivers follow a well-defined interface (through `ioctl`) allowing DIFUZE to reuse common structure across drivers. FuzzGen infers how an API is used from existing use cases and generates fuzzing functions based on observed usage. SemFuzz [69], used natural-language processing to process the CVE descriptions and extract the location of the bug. Then it uses this information to synthesize inputs that target this specific part of the vulnerable code.

Developed concurrently and independently from FuzzGen, FUDGE [70] is the most recent effort on automated fuzz driver generation. FUDGE leverages a *single* library consumer to infer valid API usages of a library to synthesize fuzzers. However there are two major differences to our approach: *First*, FUDGE extracts sequences of API calls and their context (called “snippets”) from a single library consumer and then uses these snippets to create fuzz drivers which are then tested using a dynamic analysis. Instead of extracting short snippets from consumers, FuzzGen *minimizes consumers* (iterating over the consumer’s CFG) to only the library calls, their dependent checks, and dependent arguments/data flow. *Second,* FUDGE creates many small fuzz drivers from an extracted snippet. In comparison, FuzzGen merges *multiple* consumers to a graph where sequences of arbitrary length can be synthesized. Instead of the 1-N approach of FUDGE, FuzzGen uses an M-N approach to increase flexibility. Compared to FUDGE, FuzzGen fuzzers are larger, more generic, focusing on complex API interaction and not just short API sequences.

Beside fuzzing, there are several approaches to infer API usage and specification. One way to infer API specifications [71, 72] is through dynamic analysis. This approach collects runtime traces from an application, analyzes objects and API calls and produces Finite State Machines (FSMs) that describe valid sequences of API calls. This set of API specifications is solely based on dynamic analysis. Producing rich execution traces that utilize many different aspects of the library requires the ability to generate proper inputs to the program. Similarly, API Sanitizer [73] finds violation of API usages. APISan infers correct usages of an API from other uses of the API and ranks them probabilistically, without relying on whole-program analysis. APISan leverages symbolic execution to create a database of (symbolic) execution traces and statistically infers valid API usages. APISan suffers from limited scalability due to symbolic execution. As a static analysis tool, it may result in false positives. SSLint [74] targets SSL/TLS libraries and discovers API violations based on an analyst-encoded API graph. MOPS [75] is a static analyzer that uses a set of safe programming rules and searches for violations of those rules. *Yamaguchi et. al* [76] present a technique that mines common vulnerabilities from source code, representing them as a code property graph. Based on this representation, they discover bugs in other programs.



#1: opus_packet_get_bandwidth, opus_get_version_string
#2: opus_packet_get_nb_channels, opus_get_version_string
#3: opus_decoder_create
#4: opus_decoder_ctl, opus_decoder_decode
#5: opus_decoder_decode
#6: opus_decoder_decode
#7: opus_decoder_destory
#8: opus_get_version_string

Figure 2.3.: The FuzzGen workflow. FuzzGen starts with a CFG (a) and extracts the corresponding A^2DG (b) (see (c) for the graph of another module). The two A^2DG graphs are then merged (d). The merged A^2DG is then used to create fuzzers based on function orders (e). These graphs are autogenerated by FuzzGen.

2.4 Design

To synthesize customized fuzzer stubs for a library, FuzzGen requires both the library and code that exercises the library (referred to as library *consumer*). FuzzGen leverages a whole system analysis to infer the library API, scanning consumers for library calls. The analysis detects all valid library usage, e.g., valid sequences of API calls and possible argument ranges for each call. This information is essential to create reasonable fuzzer stubs and is not available in the library itself.

By leveraging actual uses of API sequences, FuzzGen synthesizes fuzzer code that follows valid API sequences, comparable to real programs. Our library usage analysis allows FuzzGen to generate fuzzer stubs that are similar to what a human analyst would generate after *learning the API* and *learning how it is used in practice*. FuzzGen improves over a human analyst in several ways: it leverages real-world usage and builds fuzzer stubs that are close to real API invocations; it is complete and leverages all uses of a library, which could be manually overlooked; and FuzzGen scales to full systems due to its automation without requiring human interaction.

At a high level, FuzzGen consists of three distinct phases, as shown in Figure 5.1. First, FuzzGen analyzes the target library and collects all code on the system that utilizes functions from this library to infer the basic API. Second, FuzzGen builds the *Abstract API Dependence Graph* (A^2DG), which captures all valid API interactions. Third, it synthesizes fuzzer stubs based on the A^2DG.

2.4.1 Inferring the library API

FuzzGen leverages the source files from the consumers to infer the library's exported API. First, the analysis enumerates all declared functions in the target library, $\mathcal{F}_{lib}$. Then, it identifies *all* functions that are declared in all included headers of all consumers, $\mathcal{F}_{incl}$. Then, the set of potential API functions, $\mathcal{F}_{API}$ is:

$$\mathcal{F}_{API} \leftarrow \mathcal{F}_{lib} \cap \mathcal{F}_{incl} \tag{2.1}$$

FuzzGen's analysis relies on the Clang framework to extract this information during the compilation of library and consumer. To address over-approximation of inferred library functions (e.g., identification of functions that belong to another library that is used by the target library), FuzzGen applies a *progressive library inference*. Each potential API function is checked by iteratively compiling a test program linked with the target library. If linking fails, the function is not part of the library. Under-approximations are generally not a problem as functions that are exported but never used in a consumer are not reachable through attacker-controlled code.

2.4.2 A^2DG construction

FuzzGen iterates over library consumers that invoke API calls from the target library and leverages them to infer valid API interactions. It builds an abstract *layout* of library consumers which is used to construct fuzzer stubs. Recall that FuzzGen fuzzer stubs try to follow an API flow similar to that observed in real programs to build up complex state. FuzzGen fuzzer stubs allow some flexibility as some API calls may execute in random order at runtime, depending on the fuzzer's random input. The A^2DG represents the complicated interactions and dependencies between API calls, allowing the fuzzer to satisfy these dependencies. It exposes which functions are invoked first (initialization), which are invoked last (tear down), and which are dependent on each other.

The A^2DG encapsulates two types of information: *control dependencies*, and *data dependences*. Control dependencies indicate how the various API calls should be invoked, while data dependencies describe the potential dependencies between arguments and return values in the API calls (e.g., if the return value of an API call is passed as an argument in a subsequent API call).

The A^2DG is a directed graph of API calls, similar to a coarse-grained Control-Flow Graph (CFG) that expresses sequences of valid API calls in the target library. Edges are also annotated with valid parameter ranges to further improve fuzzing effectiveness as discussed in the following sections. Each node in the A^2DG corresponds to a single call of an API function, and each edge represents *control flow* between two API calls. The A^2DG encodes the control flow across the various API calls and describes which API calls are reachable from a given API call. Figure 2.3 (a) shows an instance of the CFG from a libopus consumer. The corresponding A^2DG is shown in Figure 2.3 (b).

Building the A^2DG is two step process. First, a set of *basic* A^2DGs is constructed, one A^2DG for each root function in each consumer. Second, the A^2DGs of all consumers are coalesced into a single A^2DG.

Constructing a basic A^2DG. To build a basic A^2DG, FuzzGen starts with a consumer's CFG. If the consumer is a library, FuzzGen builds CFGs for each exported API function, otherwise it starts with the `main` function. To reconcile the collection of CFGs, FuzzGen leverages the *Call Graph* of the consumer. An individual analysis starts at the entry basic block of every *root* function in the call graph to explore the full consumer. This may lead to a large number of A^2DGs for a library consumer.

Starting from the entry basic block of a root function, FuzzGen iteratively removes *every* basic block that does *not* contain any `call` instruction to an API call. If a basic block contains multiple `call` instructions on API functions, the basic block is split into multiple A^2DG nodes with one API call each. When a basic block calls a non-API function, FuzzGen recursively calculates the A^2DG for the callee and results are integrated into the caller's A^2DG. The pass integrates the calls into the root function. If the same non-API function is invoked multiple times, it is marked as a repeating function in the graph, avoiding an explosion of the graph's complexity. The algorithm to create the A^2DG is shown in algorithm 1. A call stack (C_S) prevents unbounded loops when analyzing recursive functions. Two maps (M_{entry} and M_{exit}), associate basic blocks to individual nodes in the A^2DG, allowing the algorithm to locate the A^2DG node a basic block corresponds to. Note that the only case that M_{entry} and M_{exit} are different is when a basic block contains more that one call to an API function.

After A^2DG construction, each node represents a single API call. The A^2DG allows FuzzGen to *isolate* the flows between API calls and expose their control dependencies.

Algorithm 1: A^2DG construction.

```
Input: Function $F$ to start $A^2DG$ construction
Output: The corresponding $A^2DG$
Function make_AADG(Function $F$)
    ▷ "$A \cup_= B$" is shorthand for "$A = A \cup B$"
    if $F \in C_S$ then return $(\emptyset, \emptyset)$ else $C_S \cup_= \{F\}$
    $G_{A^2DG} \leftarrow (V_{A^2DG},\ E_{A^2DG})$
    foreach basic block $B \in CFG_F$ do
        ▷ An empty vertex is not associated with an API call
        Create empty vertex $u$, $V_{A^2DG} \cup_= \{u\}$, $M_{entry}[B] \leftarrow u$
    $Q \leftarrow \{entry\_block(F)\}$    ▷ single entry point
    while $Q$ is not empty do
        remove basic block $B$ from $Q$
        $v \leftarrow M_{entry}[B]$
        foreach call instruction $c_i \in B$ in reverse order do
            if $c_i.callee \in \mathcal{F}_{API}$ then
                if $v$ is empty then
                    $v \leftarrow c_i$, $M_{entry}[B] \leftarrow v$, $M_{exit}[B] \leftarrow v$
                else
                    ▷ if already exists, split node
                    $u \leftarrow c_i$
                    $V_{A^2DG} \cup_= \{u\}$, $E_{A^2DG} \cup_= \{(u, v)\}$
                    $v \leftarrow u$, $M_{entry}[B] \leftarrow u$
            else
                $AADG' \leftarrow$ make_AADG($c_i$)
                Create empty vertex $sink$
                $V_{A^2DG} \cup_= V_{A^2DG'} \cup \{sink\}$
                $E_{A^2DG} \cup_= E_{A^2DG'}$
                foreach leaf $v_l \in AADG'$ do
                    $E_{A^2DG} \cup_= \{(v_l,\ sink)\}$
                foreach root $v_r \in AADG'$ do
                    $E_{A^2DG} \cup_= \{(v,\ v_r)\}$
        foreach unvisited successor block $B_{adj}$ of $B$ do
            add $B_{adj}$ to $Q$
            $E_{A^2DG} \cup_= \{(M_{exit}[B],\ M_{entry}[B_{adj}])\}$
    ▷ Drop empty nodes from AADG
    foreach empty node $v \in AADG$ do
        foreach predecessor $p$ of $v$ do
            foreach successor $s$ of $v$ do
                $E_{A^2DG} \cup_= \{(p, s)\}$
        remove $v$ and its edges from $V_{A^2DG}$
    $C_S \leftarrow C_S - \{F\}$
    return $G_{A^2DG}$
```

Basic A^2DG construction is a static analysis which results in some over-approximation during CFG construction due to indirect function calls. FuzzGen uses an LLVM *Link Time Optimization* (LTO) analysis pass to extract this information.

Coalescing A^2DG graphs. After generating A^2DGs for each consumer, FuzzGen merges them into a single A^2DG:

Select any two A^2DG graphs and try to coalesce them together. Repeat this process until there are no two A^2DG that can be coalesced together.

To coalesce two A^2DGs they **must** have at least one node in common. Two nodes are considered "common" if they invoke the same API call with the same arguments of the same type. FuzzGen starts from the root and selects the first common node. FuzzGen then removes the node from one graph and migrates all children, along with their sub trees, to the other A^2DG, continuously merging common nodes. A common node is a requirement, as placing the nodes from the second A^2DG at random positions will likely result in illegal target states. If there are no common nodes, FuzzGen keeps the A^2DGs separate, synthesizing two different fuzzers.

Figure 2.3 (d) shows an example of the A^2DG produced after coalescing the two A^2DGs in Figure 2.3 (b) and (c). The nodes with function `opus_decoder_destroy` are coalesced (as the argument is a handle, which has the same type), but other nodes like `opus_decoder_ctl` are not coalesced as the arguments are different. It is possible for the coalesced A^2DG to result in an *inconsistent state*, which results in an API *misuse*. That is, the coalesced A^2DG may contain a path (i.e., a subset of API calls) that violates API usage and therefore causes problems to execution state of the library. In Appendix 7.2, we explain this problem in detail.

Our experiments showed that it may be feasible to coalesce two A^2DGs without common nodes by backward-slicing and locating function usages that invoke the API call. We leave this along with other heuristics to coalesce A^2DGs into a single one, for future work.

Precision of A^2DG construction. The current FuzzGen A^2DG construction has two sources of imprecision: static analysis and merging. First, the static analysis results in an over-approximation of paths. This may result in false positives due to illegal API sequences that do not occur in real programs. Second, the merging process may over-eagerly merge two A^2DGs with different or slightly different parameters, resulting in illegal API sequences. We will discuss these sources of false positives in Section 3.8.

2.4.3 Argument flow analysis

To create effective fuzzers, the A^2DG requires both control and data dependencies. To construct the *data* dependencies between API calls FuzzGen leverages two analyses: *argument value-set inference* (what values are possible) and *argument dependence analysis* (how are individual variables reused).

Argument value-set inference. Argument value-set inference answers two questions: *which* arguments to fuzz and *how* to fuzz these arguments. Supplying arbitrary random values (i.e., “blind” fuzzing) to every argument imposes significant limitations both in the efficiency and the performance of fuzzing. Contexts, handles, and file/socket descriptors are examples that result in large numbers of false positives. Supplying random values for a descriptor in an API call results in *shallow coverage* as there are sanity checks at the beginning of the function call. Some arguments present diminishing returns when being fuzzed. Consider an argument that is used to hold output, or an argument that is part of a `switch` statement. In both cases, a fuzzer will waste cycles generating large inputs, where only a few values are meaningful. To better illustrate this, consider a fuzzer for `memcpy`:

```
void *memcpy(void *dest, const void *src, size_t n);
```

Supplying arbitrary values to `n` makes it inconsistent with the actual size of `src`, which results in a segmentation fault. However this crash does not correspond to a real bug. Also, the fuzzer may invest many cycles generating random values for the `dest` argument, which is never read by `memcpy()` (please ignore the corner case of overlapping source and destination arguments for the sake of the example).

Our analysis classifies arguments into two categories according to their type: *primitive arguments* (e.g., `char`, `int`, `float`, or `double`) and *composite arguments* (e.g., pointers, arrays, structs, or function pointers). The transitive closure of composite arguments are a collection of primitive arguments—pointers may have multiple layers (e.g., double indirect pointers), structures may contain nested structures, arrays and so on—and therefore they *cannot* be fuzzed directly. That is, they cannot be assigned a random (i.e., fuzz) value, upon the invocation of the API call but require layout-aware construction. Consider an API call that takes a pointer to an integer as the first argument. Clearly, fuzzing this argument results in segmentation faults, as the function attempts to dereference the likely invalid pointer. Instead, the pointer should point to some integer. The pointed-to address can be safely fuzzed. FuzzGen performs a data-flow analysis in the target library for every function for every argument, to infer the possible values that an argument could get.

Argument dependence analysis. Data-flow dependencies are as important as control-flow dependencies. A fuzzer must not only follow the intended sequence of API calls but must also provide matching data flow. For example, after creating a context, it must be passed to specific API calls for further processing. If this does not occur, it will likely result in a violation of a state check or a spurious memory corruption.

Data-flow dependencies to be encoded in an A^2DG can be intra-procedural and inter-procedural. First, FuzzGen identifies data dependencies through static per-function alias analysis of the code using libraries, tracking arguments and return values across API calls. Static alias analysis has the advantage of being complete, i.e., allowing any valid data-flow combinations but comes at the disadvantage of imprecision. For example, if two API calls both leverage a parameter of type `struct libcontext` then our static analysis may be unable to disambiguate if the parameters point to the same instance or to different instances. This over-approximation can result in spurious crashes. FuzzGen leverages backward and forward slicing on a per-method basis to reduce the imprecision due to basic alias analysis.

Second, FuzzGen identifies dependencies across functions: For each edge in the A^2DG, FuzzGen performs another data flow analysis for each pair of arguments and return values to infer whether whether they are dependent on each other.

Two alternative approaches could either (i) leverage concrete runtime executions of the example code which would result in an under-approximation with the challenge of generating concrete input for the runtime execution or (ii) leverage an inter-function alias analysis that would come at high analysis cost. Our approach works well in practice and we leave exploration of alternate approaches to data-flow inference as future work.

The A^2DG (i.e., API layout) exposes the order and the dependencies between the previously discovered API calls. However, the arguments for the various API calls may expose further dependencies. The task of this part is twofold: *First*, it finds dependencies between arguments. For example, if an argument corresponds to a context that is passed to multiple consecutive API calls it should likely not be fuzzed between calls. *Second*, it performs backward slicing to analyze the data flow for each argument. This gives FuzzGen some indication on how to initialize arguments.

2.4.4 Fuzzer stub synthesis

Finally, FuzzGen creates fuzzer stubs for the different API calls and its arguments through the now complete A^2DG. An important challenge when synthesizing fuzzer stubs is to balance between depth and breadth of the A^2DG exploration. For example, due to loops, a fuzzer stub could continuously call the same API function without making any progress.

Instead of generating many fuzzer stubs for each A^2DG, FuzzGen creates a single stub that leverages the fuzzer's entropy to traverse the A^2DG. At a high level, a stub encodes all possible paths (to a certain depth) through the A^2DG. The first bits of the fuzzer input encode the path through the API calls of the A^2DG. Note that FuzzGen only encodes the sequence of API calls through the bits, not the complete control flow through the library functions themselves. The intuition is that an effective fuzzer will "learn" that if certain

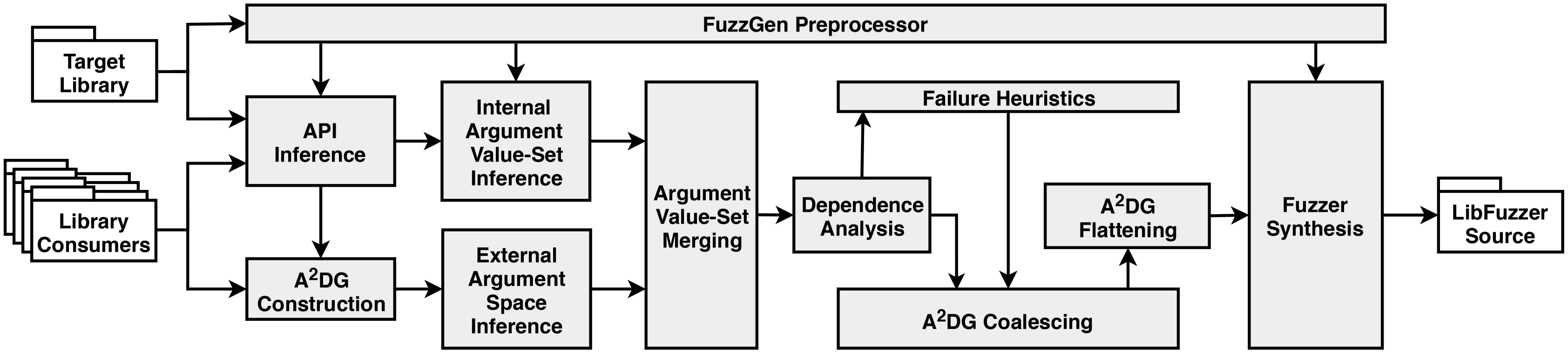


Figure 2.4.: FuzzGen implementation overview.

input encodes an interesting path, mutating later bits to explore different data-flow along that path. As soon as the path is well-explored, the fuzzer will flip bits to follow an alternate path.

2.5 Implementation

The FuzzGen prototype is written in about 19,000 lines of C++ code, consisting of LLVM/Clang [77] passes that implement the analyses and code to generate the fuzzers. FuzzGen generated fuzzers use libFuzzer [53] and are compiled with Address Sanitizer [78].

FuzzGen starts with a target library and performs a whole system analysis to discover all consumers of the library. The library and all consumers are then compiled to LLVM bitcode as our passes work on top of LLVM IR. Figure 2.4 shows a high level overview of the different FuzzGen phases.

The output of FuzzGen is a collection (one or more) of C++ source files. Each file is a fuzzer stub that utilizes libfuzzer [53] to fuzz the target library.

Target API inference. FuzzGen infers the library API by intersecting the functions implemented in the target library and those that are declared in the consumers' header files.

A^2DG **construction.** FuzzGen constructs a per-consumer A^2DG by filtering out all non-API calls from each consumer's CFG, starting from the root functions. For program consumers, the root function is `main`. To support libraries as consumers, root functions are functions with no incoming edges (using a backwards data-flow analysis to reduce the imprecision through indirect control-flow transfers).

Internal Argument Value-Set inference. Possible values and their types for the function arguments are calculated through a per-function data flow analysis. FuzzGen assigns different attributes to each argument based on these observations. These attributes allow the fuzzer to better explore the data space of the library. Note that this process is imprecise due to aliasing. Table 2.1 shows the set of possible attributes. For example, if an argument is

Attribute	**Description**
dead	Argument is not used
invariant	Argument is not modified
predefined	Argument takes a constant value from a set
random	Argument takes any (random) value
array	Argument is an array (pointers only)
array size	Argument represents an array size
output	Argument holds output (destination buffer)
by value	Argument is passed by value
NULL	Argument is a NULL pointer
function pointer	Argument is a function pointer
dependent	Argument is dependent on another argument

Table 2.1.: Set of possible attributes inferred during the argument value-set analysis.

only used in a switch statement, it can be encoded as a set of *predefined* values. Similarly, if the first access to an argument is a write, the argument is used to *output* information. Arguments that are not modified (such as file descriptors or buffer lengths) receive the *invariant* attribute.

External Argument Value-Set inference. Complementing the internal argument value-set inference, FuzzGen performs a backward slice from each API call through all consumers, assigning the same attributes to the arguments.

Argument Value-Set Merging. Due to imprecision in the analysis or potential misuses of the library, the attributes of the arguments may differ. We need to carefully consolidate the different attributes for each argument when merging the attributes. Generally, FuzzGen's analysis is more accurate with external arguments. These arguments tend to provide real use-cases of the function. Any internal assignments that give concrete values, are used to complement the externally observed values. Value-set merging is based on heuristics and may be adjusted in future work.

Dependence analysis. Knowing the possible values for each argument is not enough, the fuzzer must additionally know when to reuse the same variable across multiple functions.

The dependence analysis infers when to reuse variables and when to create new ones between function calls. FuzzGen performs a per-consumer data-flow analysis using precise intra-procedural and coarse-grained inter-procedural tracking to connect multiple API calls. While a coarse-grained inter-procedural analysis may result in imprecision, it remains tractable and scales to large consumers. The analysis records any data flow between two API functions in the A^2DG. Similarly to other steps, aliasing may lead to further imprecision.

Failure Heuristics. To handle some corner cases, FuzzGen uses a heuristic to discard error paths and dependencies. Many libraries contain ample error checking. Arguments are checked between API calls and, if an error is detected, the program signals an error. The argument analysis will detect theses checks as argument constraints. Instead of adding these checks to the A^2DG, we discard them. FuzzGen detects functions that terminate the program or pass on errors and starts the detection from there.

A^2DG **Coalescing.** After initial A^2DG construction, each consumer results in a set of at least one A^2DG. To create fuzzers that explore more state, FuzzGen tries to coalesce different A^2DG. Starting from an A^2DG node where an API call shares the exact same argument types and attributes, FuzzGen continuously merges the nodes or adds new nodes that are different. If the two graphs cannot be merged, i.e., there is a conflict for an API call then FuzzGen returns two A^2DGs. If desired, the analyst can override merging policies based on the returned A^2DGs. However, coalescing may combine an API call sequence that results in a *state inconsistency* (see Appendix 7.2 for an example). An analyst may optionally disable coalescing and produce a less generic fuzzer for each consumer. Although this approach cannot expose deeper dependencies, it increases parallelism, as different fuzzers can target different aspects of the library.

A^2DG **Flattening.** So far, the A^2DG may contain complex control flow and loops. To create simple fuzzers, we “flatten” the A^2DG before synthesizing a fuzzer. Our flattening heuristic is to traverse the A^2DG and to visit each API call at least once by removing backward edges (loops) and then applying a (relaxed) topological sort on the acyclic A^2DG

to find a valid order for API calls. While a topological sort would provide a total order of functions (and therefore result in an overly rigid fuzzer), we relax the sorting. At each step our algorithm removes all API functions of the same order and places them in a group of functions that may be called in random order.

Fuzzer Synthesis. Based on a flattened A^2DG, FuzzGen translates nodes into API calls and lays out the variables according to the inferred data flow. The fuzzer leverages some fuzz input to decode a concrete sequence for each group of functions of the same order, resulting in a random sequence at runtime. Before compiling the fuzzer, FuzzGen must also include all the necessary header files. During the consumer analysis, FuzzGen records a dependence graph of all includes and, again, uses a topological sort to find the correct order for all the header files.

FuzzGen Preprocessor. The source code to LLVM IR translation is a lossy process. To include details such as header declarations, dependencies across header files, pointer arguments, array types, argument names, and `struct` names, FuzzGen leverages a preprocessor pass that records this information for later analysis.

2.6 Evaluation

Evaluating fuzzing is challenging due to its inherent non-determinism. Even similar techniques may exhibit vastly different performance characteristics due to randomness of input generation. *Klees. et al* [5] set out guidelines and recommendations on how to properly compare different fuzzing techniques. Key to a valid comparison are (i) a sufficient number of test runs to assess the distribution using a statistical test, (ii) a sufficient length for each run, and (iii) standardized common seeds (i.e., a small set of valid corpus files in the right format).

Following these guidelines, we run our fuzzers five (5) times each (since results from a single run can be misleading), with twenty-four (24) hour timeouts. In the FuzzGen experiments, coverage tails off after a few hours with only small changes during the remainder

	Library Information						Consumer Information					Final A^2DG			
	Name	**Type**	**Src Files**	**Total LoC**	**Funcs**	**API**	**Total**	**Used**	**Total LoC**	**Avg D_c**	**UAPI**	**Graphs**	**Coal.**	**Nodes**	**Edges**
Android	libhevc	video	303	113049	314	1	2	2	3880	0.002	1	10	5	29	58
	libavc	video	190	83942	581	1	2	2	4064	0.002	1	9	4	29	53
	libmpeg2	video	118	19828	179	1	2	2	4230	0.001	1	9	5	30	56
	libopus	audio	315	50983	276	65	23	4	1079	0.074	12	4	4	24	30
	libgsm	speech	41	6145	31	8	9	4	396	0.060	7	4	4	57	88
Deb	libvpx	video	1003	352691	1210	130	40	4	594	0.075	13	4	4	29	46
	libaom	video	955	399645	4232	86	39	4	491	0.106	17	4	4	40	51

Table 2.2.: Codec libraries and consumers used in our evaluation. **Library Information**: **Src Files** = Number of source files, **Total LoC** = Total lines of code (without comments and blank lines), **Funcs** = Number of functions found in the library, **API** = Number of API functions. **Consumer Information**: **Total** = Total number of library consumers on the system, **Used** = Library consumers included in the evaluation, **Total LoC** = Total lines of code of all library consumers (without comments and blank lines), **Avg** D_c = Average consumer density, **UAPI** = Number of API functions used in the consumers. **Final** A^2DG: **Graphs** = Total number of A^2DGs, **Coalesced** = Number of nodes coalesced (same as the number of A^2DGs merges, since our algorithm uses a single node for merging), **Nodes**, **Edges** = Total number of nodes and edges (respectively) in the final A^2DG.

of the test run (see Figure 2.5). Longer timeouts appear to have a negligible effect on our results.

The effectiveness of a fuzzer depends on the number of discovered bugs. However, code coverage is a complementing metric that reflects a fuzzer's effectiveness to generate inputs that cover large portions of the program. Performance is an orthogonal factor as more executed random tests broadly increase the chances of discovering a bug.

Due to the lack of extensive previous work on library fuzzing, we cannot compare FuzzGen to other automatic library fuzzers. As mentioned in Section 3.1, the primary method for library fuzzing is to (manually) write a fuzzer stub that leverages the libFuzzer [53] engine. We evaluate our FuzzGen prototype on AOSP and Debian. Evaluating and testing FuzzGen on two different systems demonstrates the ability to operate in different environments with different sets of library consumers. Additionally, we compare FuzzGen against libFuzzer stubs written by a human analyst. A second method is to find a library consumer (which is a standalone application) and use any of the established fuzzing techniques. We forfeit the second method as the selection of the standalone application will be arbitrary and highly influence the results. There is no good metric on how an analyst would select the "best" standalone application.

To compare FuzzGen, we select seven (7) widely deployed codec libraries to fuzz. There are two main reasons for selecting codec libraries. *First*, codec libraries present a broad attack surface especially for Android, as they can be remotely reached from multiple vectors as demonstrated in the StageFright [61] attacks. *Second*, codec libraries must support a wide variety of encoding formats. They consist of complex parsing code likely to contain more bugs and vulnerabilities.

We manually analyzed the API usage of each library and wrote manual fuzzer stubs for libhevc, libavc, libmpeg2, and libgsm. Luckily AOSP already provides manually written fuzzers libopus, libvpx, and libaom which we can readily use. Some libraries such as libmpeg2 have complicated interface (see Section 2.2) and it took several weeks to sufficiently understand all libraries and write the corresponding fuzzer stubs. In comparison, FuzzGen generates a fuzzer in a few minutes given the LLVM IR of the library and the consumers.

Table 2.2 shows all libraries that we used in the evaluation for AOSP and Debian. Note that the libhevc, libavc, and libmpeg2 libraries have a single API call (see Figure 2.2 for an example) that acts as a dispatcher to a large set of internal functions. To select the appropriate operation, the program initializes a `command` field of a special `struct` which is passed to the function. Such dispatcher functions are challenging for fuzzer synthesis and we chose these libraries to highlight the effectiveness of FuzzGen.

2.6.1 Consumer Ranking

When synthesizing fuzzers, methods for consumer selection are important. Fuzzers based on more consumers tend to include more functionality. This functionality, represented by new API calls and transitions between API functions, can increase the fuzzer's complexity. An efficient fuzzer must take both the amount of API calls and the underlying complexity into account. It is important to consider how much initialization state should be constructed before fuzz input is injected into the process. It is also important to determine how many API calls should be used in a single fuzzer to target a particular aspect of the library. During the evaluation, we observed that adding certain consumers increased A^2DG complexity without increasing the API diversity or covering new functionality. Merging too many consumers increases A^2DG complexity without resulting in more interesting paths. Adding other consumers lead to the *state inconsistency* problem. Restricting the analysis to few consumers resulted in a more representative A^2DG, but lead to an interesting question: which set of consumers provide a representative set of API calls?

FuzzGen ranks the "quality" of consumers from a fuzzing perspective and creates fuzzers from high quality consumers. The intuition is the number of API calls per lines of consumer code (i.e., the fraction of API calls in a consumer) correlates to a relatively high usage of the target API. That is, FuzzGen selects consumers that are "library oriented". We empirically found that using four consumers demonstrates all features of our prototype, such as A^2DG

Library	Manual fuzzer information								FuzzGen fuzzer information								Difference		
	Total	Edge Coverage (%)				Bugs Found		exec/	Total	Edge Coverage (%)				Bugs Found		exec/	p	Cov	Bugs
	LoC	Max	Avg	Min	Std	T	U	sec	LoC	Max	Avg	Min	Std	T	U	sec			
libhevc	308	56.15	55.70	55.32	0.32	2493	23	83	1170	74.50	74.16	74.01	0.18	404	7	29	0.012	+18.46	-16
libavc	306	54.91	50.30	44.71	4.28	283	1	*8	1155	70.62	65.98	64.65	2.33	0	0	151	0.008	+15.68	-1
libmpeg2	457	51.39	49.59	45.42	2.14	1509	3	20	1204	56.95	56.60	56.26	0.26	6753	3	47	0.012	+7.01	0
libopus	125	15.85	15.71	15.16	0.27	0	0	174	624	39.99	35.22	32.63	3.08	110	3	218	0.012	+19.51	+3
libgsm	121	75.55	75.55	75.31	0.00	0	0	5966	490	69.40	68.20	67.40	0.77	229	1	4682	0.012	-7.35	+1
libvpx	122	54.79	54.13	53.61	0.49	0	0	63	481	52.17	50.99	48.05	1.52	464652	1	2060	0.012	-3.14	+1
libaom	69	44.54	35.03	30.40	5.12	57	2	111	1132	41.10	33.43	25.96	5.87	75	2	166	0.674	-1.60	0

Table 2.3.: Results from fuzzer evaluation on codec libraries. We run each fuzzer 5 times. **Total LoC** = Total lines of fuzzer code, **Edge Coverage** $\%$ = edge coverage (**max**: maximum covarage from best run, **avg**: average coverage across all runs, **min**: maximum coverage from the worst run, **std**: standard deviation of the coverage), **Bugs found** = Number of total (**T**) and unique (**U**) bugs found, **exec/sec** = Average executions per second (from all runs), **Difference** = The difference between FuzzGen and manual fuzzers (p value from Mann-Whitney U test, unique bugs and maximum edge coverage). *The executions per second in this case are low because all 283 discovered bugs are timeouts.

coalescing, and results in small fuzzers that are easy to verify. For the evaluation, the generated fuzzers are manually verified to not violate the implicit API dependencies or generate false positives.

However, in Section 2.7 we demonstrate how the number of consumers affects the set of API functions and how the generated A^2DGs participate in the fuzzer. The number of applied consumers *tail off* at a certain point. That is, additional consumers increase fuzzer complexity without adding new "interesting" coverage of the API. In future work we plan to explore other heuristics or even random selections of consumers to construct potentially more precise A^2DGs.

Formally, our heuristic for ranking consumers, is called *consumer density*, D_c, and defined as follows:

$$\mathcal{D}_c \leftarrow \frac{\#\ distinct\ API\ calls}{Total\ lines\ of\ real\ code} \tag{2.2}$$

2.6.2 Measuring code coverage

Code coverage is important both as feedback for the fuzzer during execution and to compare different fuzzers' ability to explore a given program. Code overage can be measured at different granularities: function, basic block, instruction, basic block edges, and even lines of source code. FuzzGen, like AFL and libFuzzer, uses basic block edge coverage.

For the evaluation, FuzzGen uses SanitizerCoverage [79], a feature that is available in Clang. During compilation, SanitizerCoverage adds instrumentation functions between CFG edges to trace program execution. To optimize performance, SanitizerCoverage does not add instrumentation functions on *every* edge as many edges are considered redundant. This means that the total number of edges that are available for instrumentation during fuzzing do not correspond to the total number of edges in the CFG.

Measuring code coverage for a single fuzzing run may be misleading [5]. To address this, statistical testing is conducted across the five runs to calculate the *average* coverage over time. Since new code paths are found at different times, we cannot simply calculate the average coverage for a given time. To overcome this problem we use *linear interpolation* to

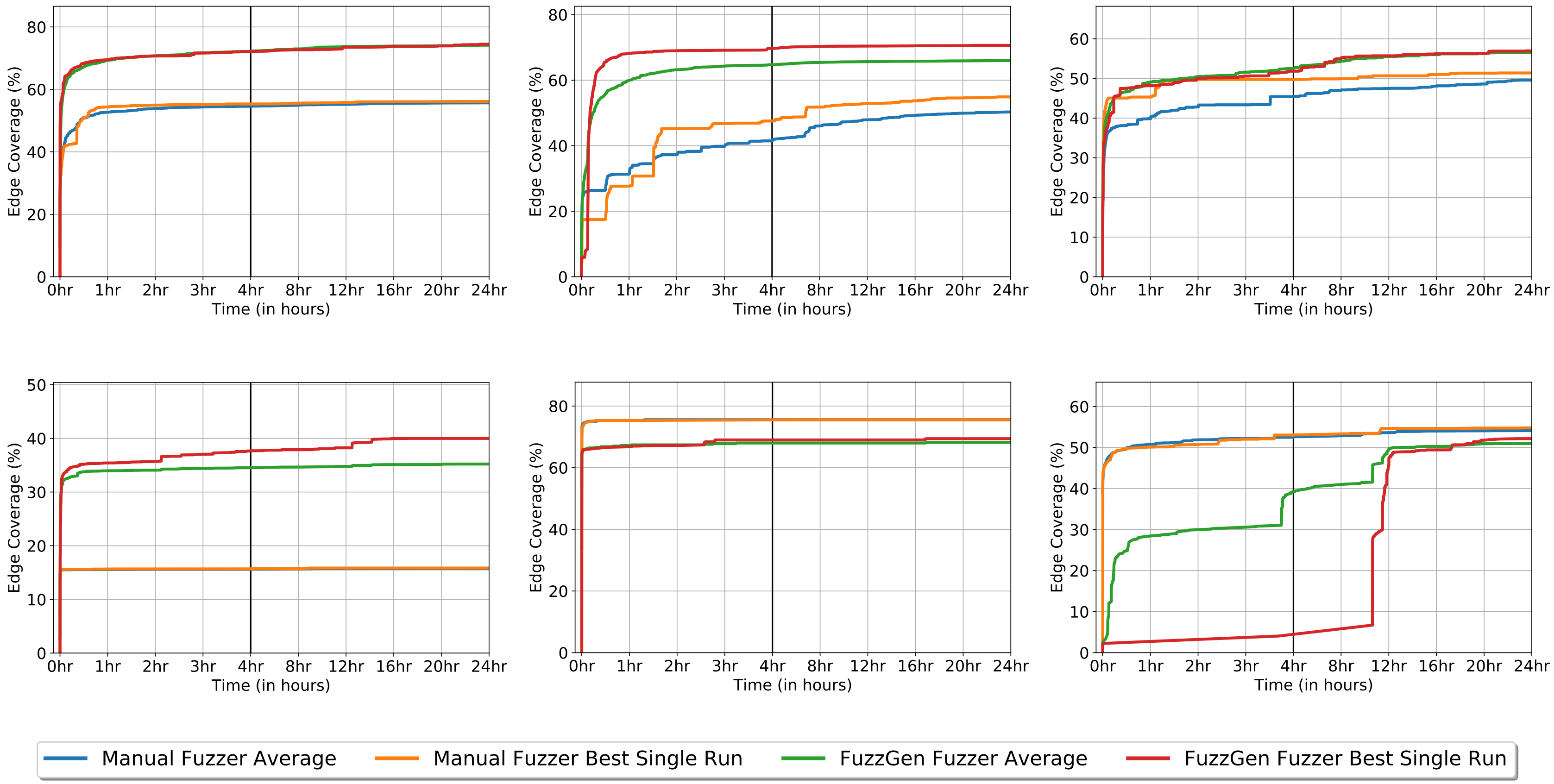


Figure 2.5.: Code coverage (%) over time for each library. The blue line shows the average edge coverage over time for manual fuzzers and the orange line shows the edge coverage for the best single run (among the five) for manual fuzzers. Similarly, the green line shows the average edge coverage for FuzzGen fuzzers, and the red line the edge coverage from best single run for FuzzGen fuzzers.

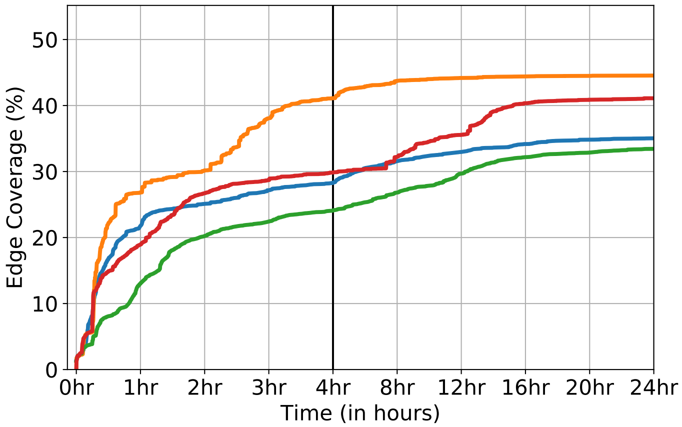


Figure 2.6.: Code coverage over time for libaom.

approximate the coverage for each fuzzing run at given time intervals. Then we calculate the average code coverage on each interval and visualize it in Figure 2.5 and Figure 2.6. Finally, we also report the results of the Mann-Whitney U test, comparing the maximum coverage across all runs for manual/generated fuzzers, according to [80]. The coverage of the FuzzGen generated fuzzers are better ($p < 0.05$, two-tailed) than the manually written fuzzers for all libraries except for libaom, where the result is not statistically significant.

2.6.3 Android evaluation

To evaluate FuzzGen on Android, we set up a small device cluster of twelve (12) Pixel-2 (walleye) devices (Figure 2.7). This setup allowed us to run repeated 24-hour fuzzing sessions and fuzz the first five (5) libraries. Table 2.3 shows the results of our fuzzing execution. The first observation is that manual fuzzers are smaller in size as they target only a specific part of the library (e.g., a decoding routine). Second, manual fuzzers are more targeted. Due to the focus on a single component, manual fuzzers can expose more bugs in that component compared to FuzzGen fuzzers. Meanwhile, FuzzGen fuzzers are broader and achieve a higher code coverage as they encode more diverse API interactions. This however imposes complexity which results in performance overhead, reducing the number

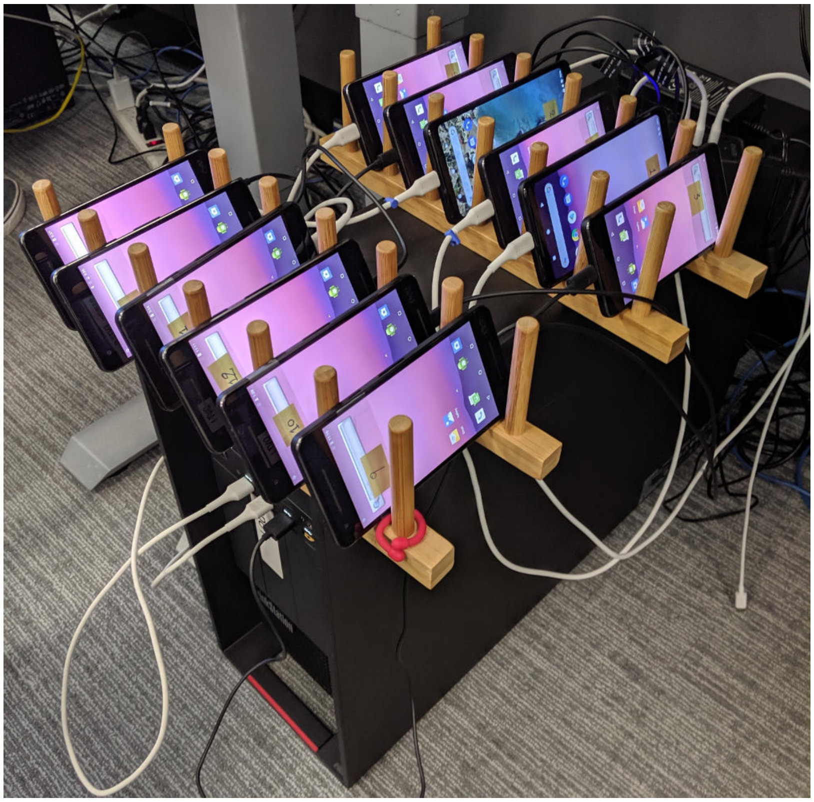


Figure 2.7.: Our device fuzzing cluster.

of executions per second. Given more additional resources, FuzzGen fuzzers therefore allow the exploration of a broader program surface in the long run.

To get a better intuition on the evolution of the fuzzing process, we visualize the edge coverage over time as shown in Figure 2.5a through Figure 2.5e. The libopus library has lower total coverage (39.99%) than the other libraries tested. This is because all selected consumers focused on the decoding functionality. This aspect is highlighted in Table 2.2, where the fuzzer includes only 12 API calls while the API exposes 65 functions. A different selection of library consumers that utilize more aspects of the library (e.g., encoding functionality), would result in higher coverage, illustrating that selection of library consumers is crucial for FuzzGen fuzzing.

One of the key advantages of FuzzGen compared to manual fuzzer collection is the scalability and automation that FuzzGen provides. FuzzGen can leverage different sets of consumers to create many different fuzzers (e.g., a fuzzer stub focusing on encoding while another fuzzer stub focuses on decoding), allowing an analyst to explore different parts of the API in depth without having to manually create individual fuzzer stubs. These automatically created fuzzers can run at scale, simplifying the work of the analyst.

2.6.4 Debian evaluation

The Debian evaluation shows similar results to the Android evaluation. Two (2) additional codec libraries were selected for FuzzGen fuzzer generation. It is important to note the difference in library consumer selection. On Android, consumers are limited to those present in AOSP. On Debian, the package manager is referenced to search for consumers that depend on the given library. In both cases we leverage a “finite” ecosystem where we can iterate through all available packages and select candidates to synthesize our fuzzers.

The last two columns of Table 2.3 show the results of running libvpx and libaom on Debian. The edge coverage over time is shown in Figure 2.5f and Figure 2.6 respectively. The first observation is that manual fuzzers have a lower rate of executions per second even though they are much smaller in size. This is because they contain *loops*. That is, they use a loop to continuously feed random input to the decoder. FuzzGen fuzzers are loop-free, which implies they spend less time on individual random inputs. For both libvpx and libaom, decoding multiple frames results in building a more complicated *decoding state*, which in turn triggers deeper code paths. It is better to have a fuzzer that contains loops for these cases, even though it achieves lower executions per second. For libopus, the decoding state is much simpler—since it is an audio library—so decoding multiple frames on each random input, affects performance, which results in a lower coverage.

2.7 Library Consumer Complexity

We empirically determined that a maximum of four consumers is a reasonable balance between complexity, breadth of explored API, and fuzzing effectiveness. To demonstrate the loss of effectiveness and the unnecessary increase in complexity when adding too may consumers, we present a case study on the `libopus` library where we continuously and

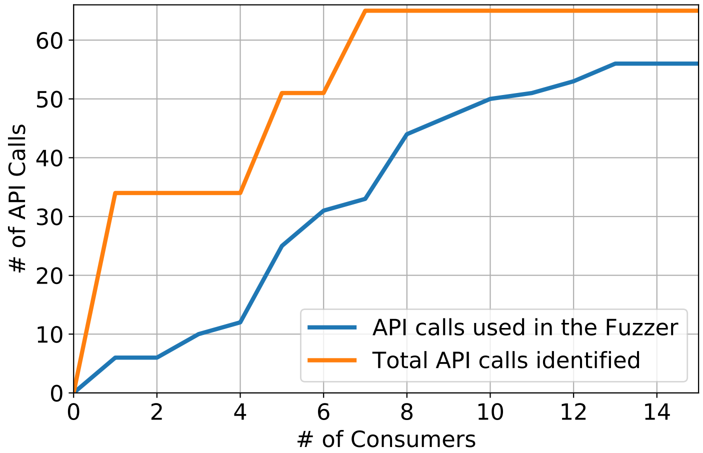


Figure 2.8.: Consumer tail off for distinct API calls for `libopus` library.

iteratively merge consumers. We start with one consumer and progressively add more consumers (following our predetermined ranking). We measure the total number of API calls found in the consumer along with the size of the corresponding A^2DG. Table 2.5 shows how the number of consumers increases the size of the explored API. Only 7 consumers are enough to discover the complete API. However, the generated fuzzer only executes 33 different API calls. With increasing number of merged consumers, the fuzzer then executes more API calls until we reach a plateau at 13 merged consumers. Note that the fuzzer creates one path through the program that strings these API calls together. Libraries expose different functionality that are hard to streamline. This additional complexity slows down the fuzzer and prohibits it from discovering bugs quickly. Additionally, the generated fuzzers are harder for an analyst to analyze and, due to the repeated merging process, we increase the chances of false positives. Our observation is that it is better to create several smaller fuzzers than one complex fuzzer.

Figure 2.8 visualizes the discovery of API calls relative to the increasing set of merged library consumers. With 15 consumers FuzzGen generates a fuzzer stub with $8,375$ lines of code. Despite this enormous size, it compiled and discovered a crash. However verifying whether this crash is a false positive or not, is extremely challenging due to the complexity of the API interactions in the fuzzer.

CVE number	Severity	Component	Description
CVE-2019-2176	Critical	libhevc	Heap Buffer Overflow in ihevcd_parse_buffering_period_sei
CVE-2019-2108	Critical	libhevc	Stack buffer overflow in ihevcd_ref_list
CVE-2019-2107	Critical	libhevc	Multiple heap buffer overflows in ihevcd_decode
CVE-2019-2106	Critical	libhevc	Stack buffer underflow in ihevc_sao_edge_offset_class2_chroma_ssse3
CVE-2017-13187	High	libhevc	Out of bounds read in ihevcd_nal_unit
CVE-2017-0858	Medium	libavc	NULL pointer dereference in ih264d_parse_decode_slice

Table 2.4.: Assigned CVEs for the vulnerabilities found by FuzzGen.

2.8 Overview of Disclosed Vulnerabilities

During our evaluation, the generated fuzzers discovered 17 vulnerabilities, 6 of which were assigned CVEs (Table 2.4). Following responsible disclosure, some vulnerabilities are still under embargo. CVE-2019-2106 [58] is a *critical* vulnerability found in the High Efficiency Video Coding (HEVC) [81] library on Android. The vulnerability is an out of bounds write—which could lead to an arbitrary write—and resides inside ihevc_sao_edge_offset_class2_chroma_ssse3, as shown below:

```
void ihevc_sao_edge_offset_class2_chroma_ssse3(UWORD8 *pu1_src,
        WORD32 src_strd, UWORD8 *pu1_src_left, UWORD8 *pu1_src_top,
        UWORD8 *pu1_src_top_left, UWORD8 *pu1_src_top_right,
        UWORD8 *pu1_src_bot_left, UWORD8 *pu1_avail,
        WORD8 *pi1_sao_offset_u, WORD8 *pi1_sao_offset_v,
        WORD32 wd, WORD32 ht) {

    WORD32 row, col;
    UWORD8 *pu1_src_top_cpy, *pu1_src_left_cpy, *pu1_src_left_cpy2;

    /* ... */

    //availability mask creation
    u1_avail0 = pu1_avail[0];
    u1_avail1 = pu1_avail[1];
    au1_mask[0] = u1_avail0;
    au1_mask[1] = u1_avail0;
    au1_mask[wd - 1] = u1_avail1;
    au1_mask[wd - 2] = u1_avail1;   // Crash. OOB write --->
```

Consumers	API		A^2DG		
	Used	Found	Total	Nodes	Edges
0	0	0	1	0	0
1	6	34	1	7	12
2	6	34	1	9	14
3	10	34	1	16	22
4	12	34	1	24	30
5	25	51	1	142	289
6	31	51	2	148	303
7	33	65	2	181	438
8	44	65	1	540	1377
9	47	65	2	551	1393
10	50	65	2	611	1473
11	51	65	2	613	1475
12	53	65	2	697	1587
13	56	65	2	883	1773
14	56	65	2	885	1778
15	56	65	2	885	1778

Table 2.5.: Complexity increase for the `libopus` library. **Consumers**: Total number of consumers used. **API**: **Used**: Total number of distinct API calls used in the final fuzzer. **Found**: Total number of distict API calls identified in headers. A^2DG: **Total**: Total number of A^2DG graphs produced (if coalescing is not possible there are more than one graphs). **Nodes** & **edges**: The total number of nodes and edges across all A^2DGs.

CVE-2017-13187 [59] is another *high severity* vulnerability found in the same component. This time, the vulnerability is an out of bounds read—which can cause remote denial of service—and resides inside `ihevcd_nal_unit` as shown below:

```
IHEVCD_ERROR_T ihevcd_nal_unit(codec_t *ps_codec)
{
    IHEVCD_ERROR_T ret = (IHEVCD_ERROR_T)IHEVCD_SUCCESS;

    /* NAL Header */
    nal_header_t s_nal;

    ret = ihevcd_nal_unit_header(&ps_codec->s_parse.s_bitstrm,
    &s_nal);

    RETURN_IF((ret != (IHEVCD_ERROR_T)IHEVCD_SUCCESS), ret);

    if(ps_codec->i4_slice_error)
        s_nal.i1_nal_unit_type =   // Crash. OOB read. --->
        ps_codec->s_parse.ps_slice_hdr->i1_nal_unit_type;
```

Supplying a frame with malformed slices to the decoder triggers both an out-of-bounds write (using the first vulnerability) and an out-of-bounds read (using the second vulnerability).

2.9 Discussion and future work

Our prototype demonstrates the power of automatic fuzzer generation and API dependency analysis. As this is a first step towards automation, we would like to highlight several opportunities for improvement.

Maximum code coverage FuzzGen generated fuzzers achieve 54.94% coverage on average compared to manually written fuzzers that achieve only 48.00%. While FuzzGen vastly simplifies the generation of fuzzers, it remains an open question if the additional coverage substantially improves the fuzzing effectiveness in itself and if full cumulative coverage can be achieved by improving FuzzGen. The coverage we report is the cumulative coverage across all inputs in a single run. Given a fuzzer stub, only a certain amount of coverage can be achieved given through a combination of the used API functions and the arguments used for those functions. The problem of the maximum possible coverage that a fuzzer can achieve given a fuzzer stub is left for future work.

Single library focus. For now, FuzzGen focuses on a single target library and does not consider interactions between libraries. FuzzGen could be extended to support multiple libraries and interactions between libraries. This extension poses the interesting challenge of complex inter-dependencies but will allow the exploration of such inter-dependencies through an automated fuzzer. We leave this as future work.

Coalescing dependence graphs into a unifying A^2DG. When multiple library consumers are available, FuzzGen has to either *coalesce* all generated A^2DG into a single one or generate a separate fuzzer of each library consumer While the first approach can expose deeper dependencies and therefore achieve deeper coverage, it could potentially result in state inconsistencies. The latter approach increases parallelism, as different clusters can fuzz different aspects of the library.

False positives. Imprecision in the static analysis and the A^2DG coalescing may result in spurious paths that result in false positives. Fuzzing libraries is inherently challenging as the API dependencies are not known. The analysis could trace benign executions and extract benign API sequences to construct the A^2DG. This would result in an under-approximation of all valid API sequences. However, the static analysis combined with A^2DG coalescing results in an over-approximation. We argue that the over-approximation results in additional freedom for the fuzzer to generate more interesting path combinations, allowing FuzzGen to trigger deep bugs at the cost of a small false positive rate. In general, we propose to validate the API sequence during triaging. The analyst can trace the set of API calls and their parameters and manually check, for each reported crash, that the API sequence is valid. We empirically discovered that for some but few merged consumers, the likelihood of false positives is low. For our evaluation, we manually verified that the fuzzers cannot create false positives by double checking all API sequences.

2.10 Conclusion

Despite their effectiveness in vulnerability discovery, existing fuzzing techniques do not transfer well to libraries. Libraries cannot run as standalone applications and fuzzing them requires either a manually written libFuzzer stub that utilizes the library, or to fuzz the library through a library consumer. The wide diversity of the API and the interface of various libraries further complicates this task. To address this challenge, we presented FuzzGen, a framework that automatically infers API interactions from a library and synthesizes a target-specific fuzzer for it. FuzzGen leverages a whole system analysis to collect library consumers and builds an Abstract API Dependence Graph (A^2DG) for them.

We evaluated FuzzGen on 7 codec libraries —which are notorious for having a complicated interface— and in all cases, the generated fuzzers were able to discover 17 previously unknown vulnerabilities and received 6 CVEs.

The source code of our prototype is available at `https://github.com/HexHive/FuzzGen`.

3 BLOCK ORIENTED PROGRAMMING: AUTOMATING DATA ONLY ATTACKS

With the widespread deployment of Control-Flow Integrity (CFI), control-flow hijacking attacks, and consequently code reuse attacks, are significantly more difficult. CFI limits control flow to well-known locations, severely restricting arbitrary code execution. Assessing the *remaining attack surface* of an application under advanced control-flow hijack defenses such as CFI and shadow stacks remains an open problem.

We introduce BOPC, a mechanism to automatically assess whether an attacker can execute arbitrary code on a binary hardened with CFI/shadow stack defenses. BOPC computes exploits for a target program from payload specifications written in a Turing-complete, high-level language called SPL that abstracts away architecture and program-specific details. SPL payloads are compiled into a program trace that executes the desired behavior on top of the target binary. The input for BOPC is an SPL payload, a starting point (e.g., from a fuzzer crash) and an arbitrary memory write primitive that allows application state corruption. To map SPL payloads to a program trace, BOPC introduces *Block Oriented Programming* (BOP), a new code reuse technique that utilizes entire basic blocks as gadgets along valid execution paths in the program, i.e., without violating CFI or shadow stack policies. We find that the problem of mapping payloads to program traces is NP-hard, so BOPC first reduces the search space by pruning infeasible paths and then uses heuristics to guide the search to probable paths. BOPC encodes the BOP payload as a set of memory writes.

We execute 13 SPL payloads applied to 10 popular applications. BOPC successfully finds payloads and complex execution traces – which would likely not have been found through manual analysis – while following the target's Control-Flow Graph under an ideal CFI policy in 81% of the cases.

3.1 Introduction

Control-flow hijacking and code reuse attacks have been challenging problems for applications written in C/C++ despite the development and deployment of several defenses. Basic mitigations include Data Execution Prevention (DEP) [8] to stop code injection, Stack Canaries [82] to stop stack-based buffer overflows, and Address Space Layout Randomization (ASLR) [19] to probabilistically make code reuse attacks harder. These mitigations can be bypassed through, e.g., information leaks [20, 83, 84, 85] or code reuse attacks [10, 11, 86, 87, 88].

Advanced control-flow hijacking defenses such as Control-Flow Integrity (CFI) [13, 89, 90, 91] or shadow stacks/safe stacks [14, 92] limit the set of allowed target addresses for indirect control-flow transfers. CFI mechanisms typically rely on static analysis to recover the Control-Flow Graph (CFG) of the application. These analyses over-approximate the allowed targets for each indirect dispatch location. At runtime, CFI checks determine if the observed target for each indirect dispatch location is within the allowed target set for that dispatch location as identified by the CFG analysis. Modern CFI mechanisms [89, 90, 93, 94] are deployed in, e.g., Google Chrome [95], Microsoft Windows 10, and Edge [96].

However, CFI still allows the attacker control over the execution along two dimensions: first, due to imprecision in the analysis and CFI's statelessness, the attacker can choose any of the targets in the set for each dispatch; second, data-only attacks allow an attacker to influence conditional branches arbitrarily. Existing attacks against CFI leverage manual analysis to construct exploits for specific applications along these two dimensions [15, 97, 98, 99, 100]. With CFI, exploits become highly program dependent as the set of reachable gadgets is severely limited by the CFI policy, so exploits must therefore follow valid paths in the CFG. Finding a path along the CFG that achieves the exploit goals is much more complex than simply finding the locations of gadgets. As a result, building attacks against advanced control-flow hijacking defenses has become a challenging, predominantly manual process.

We present BOPC (*Block Oriented Programming Compiler*) , an automatic framework to evaluate a program's remaining attack surface under strong control-flow hijacking mitigations. BOPC automates the task of finding an execution trace through a buggy program that executes arbitrary, attacker-specified behavior. BOPC compiles an “exploit” into a program trace, which is executed on top of the original program’s CFG. To express the desired exploits flexibly, BOPC provides a Turing-complete, high-level language: SPloit Language (SPL). To interact with the environment, SPL provides a rich API to call OS functions, direct access to memory, and an abstraction for hardware registers. BOPC takes as input an SPL payload and a starting point (e.g., found through fuzzing or manual analysis) and returns a trace through the program (encoded as a set of memory writes) that encodes the SPL payload.

The core component of BOPC is the mapping process through a novel code reuse technique we call *Block Oriented Programming* (BOP). First, BOPC translates the SPL payload into constraints for individual statements and, for each statement, searches for basic blocks in the target binary that satisfy these constraints (called *candidate blocks*). At this point, SPL abstracts register assignments from the underlying architecture. Second, BOPC infers a resource (register and state) mapping for each SPL statement, iterating through the set of candidate blocks and turning them into *functional blocks*. Functional blocks can be used to execute a concrete instantiation of the given SPL statement. Third, BOPC constructs a trace that connects each functional block through *dispatcher blocks*. Since the mapping process is NP-hard, to find a solution in reasonable time BOPC first prunes the set of functional blocks per statement to constrain the search space and then uses a ranking based on the proximity of individual functional blocks as a heuristic when searching for dispatcher gadgets.

We evaluate BOPC on 10 popular network daemons and setuid programs, demonstrating that BOPC can generate traces from a set of 13 test payloads. Our test payloads are both reasonable exploit payloads (e.g., calling `execve` with attacker-controlled parameters) as well as a demonstration of the computational capabilities of SPL (e.g., loops and conditionals). Applications of BOPC go beyond an attack framework. We envision BOPC as a tool for defenders and software developers to highlight the *residual* attack surface of a program.

For example, a developer can test whether a bug at a particular statement enables a practical code reuse attack in the program. Overall, we present the following contributions:

- *Abstraction:* We introduce SPL, a C dialect with access to virtual registers and an API to call OS and other library functions, suitable for writing exploit payloads. SPL enables the necessary abstraction to scale to large applications.

- *Search:* Development of a *trace module* that allows execution of an arbitrary payload, written in SPL, using the target binary's code. The trace module considers strong defenses such as DEP, ASLR, shadow stacks, and CFI alone or in combination. The trace module enables the discovery of viable mappings through a search process.

- *Evaluation:* Evaluation of our prototype demonstrates the generality of our mechanism and uncovers exploitable vulnerabilities where manual exploitation may have been infeasible. For 10 target programs, BOPC successfully generates exploit payloads and program traces to implement code reuse attacks for 13 SPL exploit payloads for 81% of the cases.

3.2 Background and Related Work

Initially, exploits relied on simple code injection to execute arbitrary code. The deployment of Data Execution Prevention (DEP) [8] mitigated code injection and attacks moved to *reusing* existing code. The first code reuse technique, *return to libc* [9], simply reused existing libc functions. *Return Oriented Programming* (ROP) [10] extended code reuse to a Turing-complete technique. ROP locates small sequences of code which end with a return instruction, called "gadgets." Gadgets are connected by injecting the correct state, e.g., by preparing a set of invocation frames on the stack [10]. A number of code reuse variations followed [11, 12, 101], extending the approach from return instructions to arbitrary indirect control-flow transfers.

Several tools [102, 103, 104, 105] seek to automate ROP payload generation. However, the automation suffers from inherent limitations. These tools fail to find gadgets in the target binary that do not follow the expected form “`inst1; inst2; ... retn;`” as they search for a set of hard coded gadgets that form pre-determined gadget chains. Instead of abstracting the required computation, they search for specific gadgets. If any gadget is not found or if a more complex gadget chain is needed, these tools degenerate to gadget dump tools, leaving the process of gadget chaining to the researcher who manually creates exploits from discovered gadgets.

The invention of code reuse attacks resulted in a plethora of new detection mechanisms based on execution anomalies and heuristics [106, 107, 108, 109, 110] such as frequency of return instructions. Such heuristics can often be bypassed [111].

While the aforementioned tools help to craft appropriate payloads, finding the vulnerability is an orthogonal process. Automatic Exploit Generation (AEG) [6] was the first attempt to automatically find vulnerabilities and generate exploits for them. AEG is limited in that it does not assume any defenses (such as the now basic DEP or ASLR mitigations). The generated exploits are therefore buffer overflows followed by static shellcode.

3.2.1 Control Flow Integrity

Control Flow Integrity [13, 89, 90, 91] (CFI) *mitigates* control-flow hijacking to arbitrary locations (and therefore code reuse attacks). CFI restricts the set of potential targets that are reachable from an indirect dispatch. While CFI does not stop the initial memory corruption, it validates the code pointer before it is used. CFI infers an (overapproixmate) CFG of the program to determine the allowed targets for each indirect control-flow transfer. Before each indirect dispatch, the target address is checked to determine if it is a valid edge in the CFG, and if not an exception is thrown. This limits the freedom for the attacker, as she can only target a small set of targets instead of any executable byte in memory. For example, an attacker may overwrite a function pointer through a buffer overflow, but the function pointer is checked before it is used. Note that CFI targets *forward edges*, i.e., virtual dispatchers for C++ or indirect function calls for C.

With CFI, code reuse attacks become harder, but not impossible [15, 97, 98, 99]. Depending on the application and strength of the CFI mechanism, CFI can be bypassed with Turing-complete payloads, which are often highly complex to comply with the CFG. So far, these code-reuse attacks rely on manually constructed payloads.

Deployed CFI implementations [89, 90, 93, 94, 112] use a static over-approximation of the CFG based on method prototypes and class hierarchy. PittyPat [113] and PathArmor [114] introduce path sensitivity that evaluates partial execution paths. Newton [115] introduced a framework that reasons about the strength of defenses, including CFI. Newton exposes indirect pointers (along with their allowed target set) that are reachable (i.e., controllable by an adversary) through given entry points. While Newton displays all usable "gadgets," it cannot stitch them together and effectively is a CFI-aware ROP gadget search tool that helps an analyst to manually construct an attack.

3.2.2 Shadow Stacks

While CFI protects *forward* edges in the CFG (i.e., function pointers or virtual dispatch), a shadow stack orthogonally protects *backward* edges (i.e., return addresses). Shadow stacks keep a protected copy (called *shadow*) of all return addresses on a separate, protected stack. Function calls store the return address both on the regular stack and on the shadow stack. When returning from a function, the mitigation checks for equivalence and reports an error if the two return addresses do not match. The shadow stack itself is assumed to be at a protected memory location to keep the adversary from tampering with it. Shadow stacks enforce stack integrity and protect the binary from any control-flow hijacking attack against the backward edge.

3.2.3 Data-only Attacks

While CFI mitigates code-reuse attacks, CFI cannot stop data-only attacks. Manipulating a program's *data* can be enough for a successful exploitation. Data-only attacks target the program's data rather than its control flow. E.g., having full control over the arguments to `execve()` suffices for arbitrary command execution. Also, data in a program may be sensitive: consider overwriting the `uid` or a variable like `is_admin`. *Data Oriented Programming* (DOP) [16] is the generalization of data-only attacks. Existing DOP attacks rely on an analyst to identify sensitive variables for manual construction.

Similarly to CFI, it is possible to build the *Data Flow Graph* of the program and apply *Data Flow Integrity* (DFI) [18] to it. However, to the best of our knowledge, there are no practical DFI-based defenses due to prohibitively high overhead of data-flow tracking.

In comparison to existing data-only attacks, BOPC automatically generates payloads based on a high-level language. The payloads follow the valid CFG of the program but not its Data Flow Graph.

3.3 Assumptions and Threat Model

Our threat model consists of a binary with a known memory corruption vulnerability that is protected with the state-of-the-art control-flow hijack mitigations, such as CFI along with a Shadow Stack. Furthermore, the binary is also hardened with DEP, ASLR and Stack Canaries.

We assume that the target binary has an arbitrary memory write vulnerability. That is, the attacker can write *any* value to *any* (writable) address. We call this an *Arbitrary memory Write Primitive* (AWP). To bypass probabilistic defenses such as ASLR, we assume that the attacker has access to an information leak, i.e., a vulnerability that allows her to read *any* value from *any* memory address. We call this an *Arbitrary memory Read Primitive* (ARP). Note that the ARP is optional and only needed to bypass orthogonal probabilistic defenses.

We also assume that there exists an entry point, i.e., a location that the program reaches naturally after completion of all AWPs (and ARPs). Thus BOPC does *not* require code pointer corruption to reach the entry point. Determining an entry point is considered to be part of the vulnerability discovery process. Thus, finding this entry point is orthogonal to our work.

Note that these assumptions are in line with the threat model of control-flow hijack mitigations that aim to prevent attackers from exploiting arbitrary read and write capabilities. These assumptions are also practical. Orthogonal bug finding tools such as fuzzing often discover arbitrary memory accesses that can be abstracted to the required arbitrary read and writes, placing the entry point right after the AWP. Furthermore, these assumptions map to real bugs. Web servers, such as nginx, spawn threads to handle requests and a bug in the request handler can be used to read or write an arbitrary memory address. Due to the request-based nature, the adversary can repeat this process multiple times. After the completion of the state injection, the program follows an alternate and disjoint path to trigger the injected payload.

These assumptions enable BOPC to inject a payload into a target binary's address space, modifying its memory state to execute the payload. BOPC assumes that the AWP (and/or ARP) may be triggered multiple times to modify the memory state of the target binary. After the state modification completes, the SPL payload executes without using the AWP (and/or ARP) further. This separates SPL execution into two phases: state modification and payload execution. The AWP allows state modification, BOPC infers the required state change to execute the SPL payload.

3.4 Design

Figure 3.1 shows how BOPC automates the analysis tasks necessary to leverage AWPs to produce a useful exploit in the presence of strong defenses, including CFI. First, BOPC

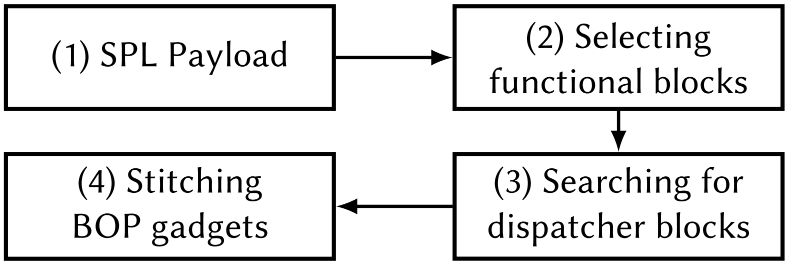

(1) SPL Payload
(2) Selecting functional blocks
(3) Searching for dispatcher blocks
(4) Stitching BOP gadgets

Figure 3.1.: Overview of BOPC's design.

provides an exploit programming language, called SPL, that enables analysts to define exploits independent of the target program or underlying architecture. Second, to automate SPL gadget discovery, BOPC finds basic blocks from the target program that implement individual SPL statements, called *functional blocks*. Third, to chain basic blocks together in a manner that adheres with CFI and shadow stacks, BOPC searches the target program for sequences of basic blocks that connect pairs of neighboring functional blocks, which we call *dispatcher blocks*. Fourth, BOPC simulates the BOP chain to produce a payload that implements that SPL payload from a chosen AWP.

The BOPC design builds on two key ideas: *Block Oriented Programming* and *Block Constraint Summaries*. First, defenses such as CFI impose stringent restrictions on transitions between gadgets, so an exploit no longer has the flexibility of setting the instruction pointer to arbitrary values. Instead, BOPC implements *Block Oriented Programming* (BOP), which constructs exploit programs called *BOP chains* from basic block sequences in the valid CFG of a target program. Note that our CFG encodes both *forward edges* (protected by CFI) and *backward edges* (protected by shadow stack). For BOP, gadgets are chains of *entire* basic blocks (sequences of instructions that end with a direct *or* indirect control-flow transfer), as shown in Figure 3.2. A BOP chain consists of a sequence of *BOP gadgets* where each BOP gadget is: one *functional block* that implements a statement in an SPL payload and zero or more *dispatcher blocks* that connect the functional block to the next BOP gadget in a manner that complies with the CFG.

Second, BOPC abstracts each basic block from individual instructions into *Block Constraint Summaries*, enabling blocks to be employed in a variety of different ways. That is, a single block may perform multiple functional and/or dispatching operations by utilizing different sets of registers for different operations. That is, a basic block that modifies a register in a manner that may fulfill an SPL statement may be used as a functional block, otherwise it may be considered to serve as a dispatcher block.

BOPC leverages abstract Block Constraint Summaries to apply blocks in multiple contexts. At each stage in the development of a BOP chain, the blocks that may be employed next in the CFG as dispatcher blocks to connect two functional blocks depend on the block

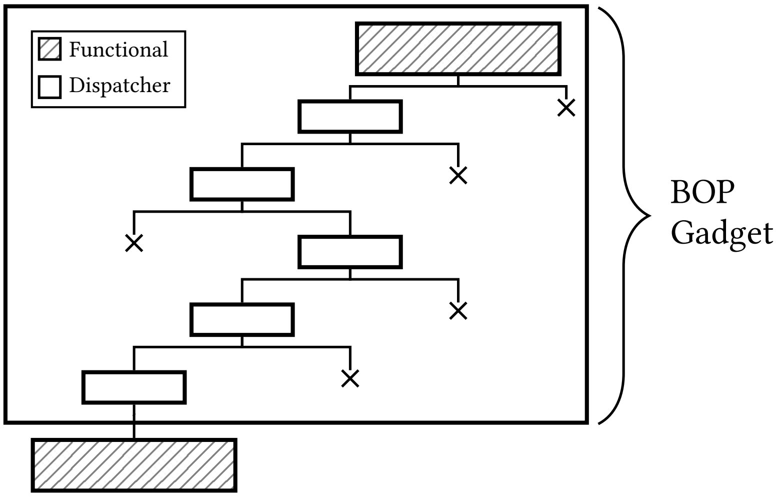


Figure 3.2.: BOP gadget structure. The *functional* part consists of a single basic block that executes an SPL statement. Two functional blocks are chained together through a series of *dispatcher* blocks, without clobbering the execution of the previous functional blocks.

summary constraints for each block. There are two cases: either the candidate dispatcher block's summary constraints indicate that it will modify the register state set and/or the memory state by the functional blocks, called the *SPL state*, or it will not, enabling the computation to proceed without disturbing the effects of the functional blocks. A block that modifies a current SPL state unintentionally, is said to be a *clobbering block* for that state. Block summary constraints enable identification of clobbering blocks at each point in the search.

An important distinction between BOP and conventional ROP (and variants) is that the problem of computing BOP chains is NP-hard, as proven in Section 7.4. Conventional ROP assumes that indirect control-flows may target any executable byte in memory while BOP must follow a legal path through the CFG for any chain of blocks, resulting in the need for automation.

Table 3.1.: Examples of SPL payloads.

Simple loop	*Spawn a shell*
void payload() {	void payload() {
__r0 = 0;	string prog = "/bin/sh\0";
	int64 *argv = {&prog, 0x0};
LOOP:	
__r0 += 1;	__r0 = &prog;
if (__r0 != 128)	__r1 = &argv;
goto LOOP;	__r2 = 0;
returnto 0x446730;	execve(__r0, __r1, __r2);
}	}

3.4.1 Expressing Payloads

BOPC provides a programming language, called *SPloit Language* (SPL) that allows analysts to express exploit payloads in a compact high-level language that is independent of target programs or processor architectures. SPL is a dialect of C. Compared to minDOP [16], SPL allows use of both virtual registers and memory for operations and declaration of variables/constants. Table 3.1 shows some sample payloads. Overall, SPL has the following features:

- It is Turing-complete;
- It is architecture independent;
- It is close to a well known, high level language.

Compared to existing exploit development tools [102, 103, 104], the architecture independence of SPL has important advantages. First, the same payload can be executed under different ISAs or operating systems. Second, SPL uses a set of *virtual registers*, accessed through reserved volatile variables. Virtual registers increase flexibility, which in turn increases the chances of finding a solution: virtual registers may be mapped to any general purpose register and the mapping may be changed dynamically.

To interact with the environment, SPL defines a concise API to access OS functionality. Finally, SPL supports conditional and unconditional jumps to enable control-flow transfers to arbitrary locations. This feature makes SPL a Turing-complete language, as proven in Appendix 7.5. The complete language specifications are shown in Appendix 7.3 in Extended Backus–Naur form (EBNF).

The environment for SPL differs from that of conventional languages. Instead of running code directly on a CPU, our compiler encodes the payload as a mapping of instructions to functional blocks. That is, the underlying runtime environment is the target binary and its program state, where payloads are executed as side effects of the underlying binary.

3.4.2 Selecting functional blocks

To generate a BOP chain for an SPL payload, BOPC must find a sequence of blocks that implement each statement in the SPL payload, which we call *functional blocks*. The process of building BOP chains starts by identifying functional blocks per SPL statement.

Conceptually, BOPC must compare each block to each SPL statement to determine if the block can implement the statement. However, blocks are in terms of machine code and SPL statements are high-level program statements. To provide flexibility for matching blocks to SPL statements, BOPC computes *Block Constraint Summaries*, which define the possible impacts that the block would have on SPL state. Block Constraint Summaries provide flexibility in matching blocks to SPL statements because there are multiple possible mappings of SPL statements and their virtual registers to the block and its constraints on registers and state.

The constraint summaries of each basic block are obtained by *isolating* and *symbolically* executing it. The effect of symbolically executing a basic block creates a set of constraints, mapping input to the resultant output. Such constraints refer to registers, memory locations, jump types and external operations (e.g., library calls).

To find a match between a block and an SPL statement the block must perform all the operations required for that SPL statement. More specifically, the constraints of the basic block should contain all the operations required to implement the SPL statement.

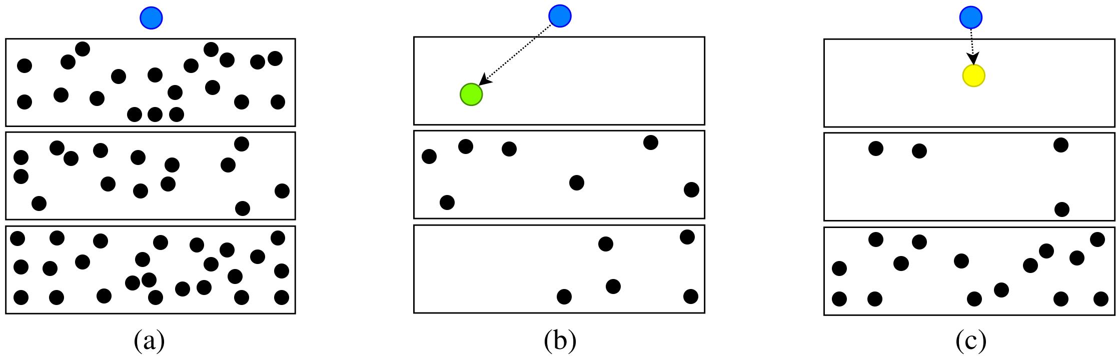


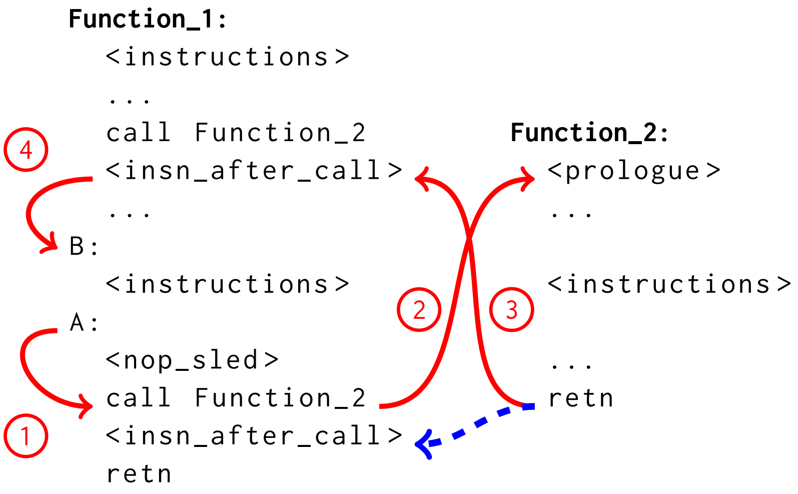
Figure 3.3.: Visualisation of BOP gadget volatility, rectangles: SPL statements, dots: functional blocks (a). Connecting any two statements through dispatcher blocks constrains remaining gadgets (b), (c).

3.4.3 Finding BOP gadgets

BOPC computes a *set* of all *potential* functional blocks for each SPL statement or halts if any statement has no blocks. To stitch functional blocks, BOPC must select one functional block and a sequence of dispatcher blocks that reach the next functional block in the payload. The combination of a functional block and its dispatcher blocks is called a *BOP gadget*, as shown in Figure 3.2. To build a BOP gadget, BOPC must select *exactly* one functional block from each set and find the appropriate dispatcher blocks to connect to a subsequent functional block.

However, dispatcher paths between two functional blocks may not exist either because there is no legal path in the CFG between them, or the control flow cannot reach the next block due to unsatisfiable runtime constraints. This constraint imposes limits on functional block selection, as the existence of a dispatcher path depends on the *previous* BOP gadgets.

BOP gadgets are *volatile*: gadget feasibility changes based on the selection of prior gadgets for the target binary. This is illustrated in Figure 3.3. The problem of selecting a suitable sequence of functional blocks, such that a dispatcher path exists between every possible control flow transfer in the SPL payload, is NP-hard, as we prove in Appendix 7.4. Even worse, an approximation algorithm does not exist.



```
Function_1:
    <instructions>
    ...
    call Function_2          Function_2:
    <insn_after_call>            <prologue>
    ...                          ...
B:
    <instructions>               <instructions>
A:
    <nop_sled>                   ...
    call Function_2              retn
    <insn_after_call>
    retn
```

Figure 3.4.: Existing shortest path algorithms are unfit to measure proximity in the CFG. Consider the shortest path from A to B. A context-unaware shortest path algorithm will mark the red path as solution, instead of following the blue arrow upon return from Function_2, it follows the red arrow (3).

As the problem is unsolvable in polynomial time in the general case, we propose several heuristics and optimizations to find solutions in reasonable amounts of time. BOPC leverages basic block *proximity* as a metric to “rank” dispatcher paths and organizes this information into a special data structure, called a *delta graph* that provides an efficient way to probe potential sequences of functional blocks.

3.4.4 Searching for dispatcher blocks

While each functional block executes a statement, BOPC must chain multiple functional blocks together to execute the SPL payload. Functional blocks are connected through zero or more basic blocks that do not clobber the SPL state computed thus far. Finding such non-clobbering blocks that transfer control from one functional statement to another is challenging as each additional block increases the constraints and path dependencies. Thus, we propose a graph data structure, called the *delta graph*, to represent the state of the search for dispatcher blocks. The delta graph stores, for each functional block for each SPL statement, the shortest path to the next candidate block. Stitching arbitrary sequences of statements is NP-hard as each selected path between two functional statements influences the availability of further candidate blocks or paths, we therefore leverage the delta graph to try *likely* candidates first.

The intuition behind the *proximity* of functional blocks is that shorter paths result in simpler and more likely satisfiable constraints. Although this metric is a *heuristic*, our evaluation (Section 3.6) shows that it works well in practice.

The delta graph enables quick elimination of sets of functional blocks that are highly unlikely to have dispatcher blocks and thus constitute a BOP gadget. For instance, if there is no valid path in the CFG between two functional blocks (e.g., if execution has to traverse the CFG "backwards"), no dispatcher will exist and therefore, these two functional blocks cannot be part of the solution.

The delta graph is a multi-partite, directed graph that has a set of functional block nodes for *every* payload statement. An edge between two functional blocks represents the *minimum* number of executed basic blocks to move from one functional block to the other, while *avoiding* clobbering blocks. See Figure 3.7 for an example.

Indirect control-flow transfers pose an interesting challenge when calculating the shortest path between two basic blocks in a CFG: while they statically allow multiple targets, at runtime they are context sensitive and only have one *concrete* target.

Our context-sensitive shortest path algorithm is a *recursive* version of Dijkstra's [116] shortest path algorithm that *avoids* all clobbering blocks.. Initially, each edge on the CFG has a cost of 1. When it encounters a basic block with a call instruction, it recursively calculates the shortest paths starting from the calling function's entry block, B_E (a *call stack* prevents deadlocks for recursive callees). If the destination block, B_D, is inside the callee, the shortest path is the concatenation of the two individual shortest paths from the beginning to B_E and from B_E to B_D. Otherwise, our algorithm finds the *shortest path* from the B_E to the closest return point and uses this value as an edge weight for that callee.

After creation of the delta graph, our algorithm selects *exactly* one node (i.e., functional block) from each set (i.e., payload statement), to *minimize* the total weight of the resulting

Table 3.2.: A counterexample that demonstrates why proximity between two functional blocks can be inaccurate. Left, we can move from point A to point B even if they are 5 blocks apart from each other. Right, it is much harder to satisfy the constrains and to move from A to B, despite the fact that A and B are only 1 block apart.

Long path with simple constraints	*Short path with complex constraints*
`a, b, c, d, e = input();`	`a = input();`
`// point A`	
`if (a == 1) {`	`X = sqrt(a);`
`  if (b == 2) {`	`Y = log(a*a*a - a)`
`    if (c == 3) {`	
`      if (d == 4) {`	`// point A`
`        if (e == 5) {`	`if (X == Y) {`
`          // point B`	`  // point B`

induced subgraph [1]. This selection of functional blocks is considered to be the most likely to give a solution, so the next step is to find the exact dispatcher blocks and create the BOP gadgets for the SPL payload.

3.4.5 Stitching BOP gadgets

The minimum induced subgraph from the previous step determines a set of functional blocks that may be stitched together into an SPL payload. This set of functional blocks has minimal distance to each other, thus making satisfiable dispatcher paths more likely.

To find a dispatcher path between two functional blocks, BOPC leverages *concolic execution* [117] (symbolic execution along a given path). Along the way, it collects the required constraints that are needed to lead the execution to the next functional block. Symbolic execution engines [118, 119] translate basic blocks into sets of constraints and use Satisfiability Modulo Theories (SMT) to find satisfying assignments for these constraints; symbolic execution is therefore NP-complete. Starting from the (context sensitive) shortest path between the functional blocks, BOPC *guides* the symbolic execution engine, collecting the corresponding constraints.

[1] The *induced subgraph* of the delta graph is a subgraph of the delta graph with one node (functional block) for each SPL statement and with edges that represent their shortest available dispatcher block chain.

To construct an SPL payload from a BOP chain, BOPC launches concolic execution from the first functional block in the BOP chain, starting with an empty state. At each step BOPC tries the first K shortest dispatcher paths until it finds one that reaches the next functional block (the edges in the minimum induced subgraph indicate which is the "next" functional block). The corresponding constraints are added to the current state. The search therefore *incrementally* adds BOP gadgets to the BOP chain. When a functional block represents a conditional SPL statement, its node in the induced subgraph contains two outgoing edges (i.e., the execution can transfer control to two different statements). However during the concolic execution, the algorithm does not know which one will be followed, it *clones* the current state and independently follows both branches, exactly like symbolic execution [118].

Reaching the last functional block, BOPC checks whether the constraints have a satisfying assignment and forms an exploit payload. Otherwise, it falls back and tries the next possible set of functional blocks. To repeat that execution on top of the target binary, these constraints are concretized and translated into a memory layout that will be initialized through AWP in the target binary.

3.5 Implementation

Our open source prototype, BOPC, is implemented in Python and consists of approximately $14,000$ lines of code. The current prototype focuses on x64 binaries, we leave the (straightforward) extension to other architectures such as x86 or ARM as future work. BOPC requires three distinct inputs:

- The exploit payload expressed in SPL,
- The vulnerable application on top of which the payload runs,
- The entry point in the vulnerable application, which is a location that the program reaches naturally and occurs after all AWPs have been completed.

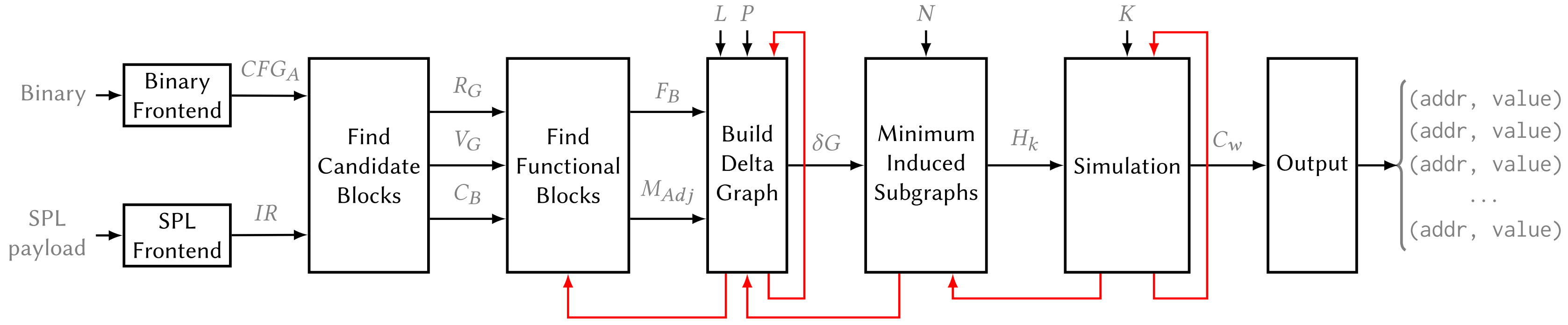


Figure 3.5.: High level overview of the BOPC implementation. The red arrows indicate the iterative process upon failure. CFG_A: CFG with basic block abstractions added, IR: Compiled SPL payload R_G: Register mapping graph, V_G: All variable mapping graphs, C_B: Set of candidate blocks, F_B: Set of functional blocks, M_{Adj}: Adjacency matrix of SPL payload, δG: Delta graph, H_k: Induced subgraph, C_w: Constraint set. L: Maximum length of continuous dispatcher blocks, P: Upper bound on payload "shuffles", N: Upper bound on minimum induced subgraphs, K: Upper bound on shortest paths for dispathers.

The output of BOPC is a sequence of $(address, value, size)$ tuples that describe how the memory should be modified during the state modification phase (Section 3.3) to execute the payload. Optionally, it may also generate some additional $(stream, value, size)$ tuples that describe what additional input should be given on any potentially open "streams" (file descriptors, sockets, stdin) that the attacker controls during the execution of the payload.

A high level overview of BOPC is shown in Figure 3.5 (a detailed implementation overview is shown in Appendix 7.7). Our algorithm is *iterative*; that is, in case of a failure, the red arrows, indicate which module is executed next.

3.5.1 Binary Frontend

The Binary Frontend uses angr [119] to lift the target binary into the VEX intermediate representation to expose the application's CFG. Operating directly on basic blocks is cumbersome and heavily dependent on the Application Binary Interface (ABI). Instead, we translate each basic block into a *block constraint summary*. Abstraction leverages symbolic execution [2] to "summarize" the basic block into a set of constraints encoding changes in registers and memory, and any potential system, library call, or conditional jump at the end of the block – generally any *effect* that this block has on the program's state. BOPC executes each basic block in an isolated environment, where every action (such as accesses to registers or memory) is monitored. Therefore, instead of working with the instructions of each basic block, BOPC utilizes its abstraction for all operations. The abstraction information for every basic block is added to the CFG, resulting in CFG_A.

3.5.2 SPL Frontend

The SPL Front end translates the exploit payload into a graph-based Intermediate Representation (IR) for further processing. To increase the flexibility of the mapping process, statements in a sequence can be executed out-of-order. For each statement sequence we build a *dependence graph* based on a customized version of Kahn's topological sorting algorithm [120], to infer all groups of independent statements. Independent statements in a subsequence are then turned into a set of statements which can be executed out-of-order. This results in a set of equivalent payloads that are permutations of the original. Our goal is to find a solution for *any* of them.

3.5.3 Locating candidate block sets

SPL is a high level language that hides the underlying ABI. Therefore, BOPC looks for potential ways to "map" the SPL environment to the underlying ABI. The key insight in this step is to find all possible ways to map the individual elements from the SPL environment to the ABI (though *candidate blocks*) and then iteratively selecting valid subsets from the ABI to "simulate" the environment of the SPL payload.

Once the CFG_A and the IR are generated, BOPC searches for and marks *candidate* basic blocks, as described in Section 3.4.2. For a block to be a candidate, it must "semantically match" with one (or more) payload statements. Table 3.3 shows the matching rules. Note that variable assignments, unconditional jumps, and returns do not require a basic block and therefore are excluded from the search.

All statements that assign or modify registers require the basic block to apply the same operation on the same, as yet undetermined, hardware registers. For function calls, the requirement for the basic block is to invoke the same call, either as a system call or as a library call (if the arguments are different, the block is clobbering). Note that the calling convention exposes the register mapping.

Upon a successful matching, BOPC builds the following data structures:

- R_G, the *Register Mapping Graph* which is a bipartite undirected graph. The nodes in the two sets represent the virtual and hardware registers respectively. The edges represent potential associations between virtual and hardware registers.

Table 3.3.: Semantic matching of SPL statements to basic blocks. **Abstraction** indicates the requirements that the basic block abstraction needs to have to match the SPL statement in the **Form**. Upon a match, the appropriate **Actions** are taken. $\_r_\alpha$, $\_r_\beta$: Virtual registers, reg_γ, reg_δ: Hardware registers, C: Constant value, V: SPL variable, A: Memory address, R_G: Register mapping graph, V_G: Variable mapping graph, D_M: Dereferenced Addresses Set, `Ijk_Call`: A call to an address, `Ijk_Boring`: A normal jump to an address.

Statement	Form	Abstraction	Actions			Example
Register Assignment	$\_r_\alpha = C$	$reg_\gamma \leftarrow C$	$R_G \cup \{(\_r_\alpha, reg_\gamma)\}$	–		`movzx rax, 7h`
		$reg_\gamma \leftarrow *A$		$D_M \cup \{A\}$		`mov rax, ds:fd`
	$\_r_\alpha = \&V$	$reg_\gamma \leftarrow C,\ C \in R \wedge W$		$V_G^{\alpha\gamma} \cup \{(V, A)\}$	–	`lea rcx, [rsp+20h]`
		$reg_\gamma \leftarrow *A$			$D_M \cup \{A\}$	`mov rdx, [rsi+18h]`
Register Modification	$\_r_\alpha \odot= C$	$reg_\gamma \leftarrow reg_\gamma \odot C$	$R_G \cup \{(\_r_\alpha, reg_\gamma)\}$			`dec rsi`
Memory Read	$\_r_\alpha = *\_r_\beta$	$reg_\gamma \leftarrow *reg_\delta$	$R_G \cup \{(\_r_\alpha, reg_\gamma),\ (\_r_\beta, reg_\delta)\}$			`mov rax, [rbx]`
Memory Write	$*\_r_\alpha = \_r_\beta$	$*reg_\gamma \leftarrow reg_\delta$				`mov [rax], [rbx]`
Call	$call(\_r_\alpha,\ \_r_\beta,\ ...)$	`Ijk_Call` to *call*	$R_G \cap \{(\_r_\alpha, \%rdi),\ (\_r_\beta, \%rsi),\ ...\}$			`call execve`

Conditional Jump	$if\ (\_r_\alpha \odot= C)$
	$goto\ LOC$
		`Ijk_Boring` $\wedge$
		$condition = reg_\gamma \odot C$
			$R_G \cup \{(\_r_\alpha, reg_\gamma)\}$			`test rax, rax`
						`jnz LOOP`

- V_G, the *Variable Mapping Graph*, which is very similar to R_G, but instead associates payload variables to underlying memory addresses. V_G is unique for every edge in R_G i.e.:

$$\forall(\_r_\alpha, reg_\gamma) \in R_G \; \exists! \, V_G^{\alpha\gamma}$$

- D_M, the *Memory Dereference Set*, which has all memory addresses that are dereferenced and their values are loaded into registers. Those addresses can be symbolic expressions (e.g., `[rbx + rdx*8]`), and therefore we do not know the concrete address they point to until execution reaches them (see Section 3.5.6).

After this step, each SPL statement has a *set* of candidate blocks. Note that a basic block can be candidate for multiple statements. If for some statement there are no candidate blocks, the algorithm halts and reports that the program cannot be synthesized.

3.5.4 Identifying functional block sets

After determining the set of candidate blocks, C_B, BOPC *iteratively* identifies, for each SPL statement, which candidate blocks can serve as functional blocks, i.e., the blocks that perform the operations. This step determines for each candidate block if there is a resource mapping that satisfies the block's constraints.

BOPC identifies the *concrete* set of hardware registers and memory addresses that execute the desired statement. A successful mapping identifies candidate blocks that can serve as functional blocks.

To find the hardware-to-virtual register association, BOPC searches for a *maximum bipartite matching* [116] in R_G. If such a mapping does not exist, the algorithm halts. The selected edges indicate the set of V_G graphs that are used to find the memory mapping, i.e., the variable-to-address association (see Section 3.5.3, there can be a V_G for every edge in R_G). Then for every V_G the algorithm repeats the same process to find another maximum bipartite matching.

This step determines, for each statement, which concrete registers and memory addresses are reserved. Merging this information with the set of candidate blocks constructs each block's SPL state, enabling the removal of candidate blocks that are unsatisfiable.

However, there may be multiple candidate blocks for each SPL statement, and thus the maximum bipartite match may not be unique. The algorithm enumerates *all* maximum bipartite matches [121], trying them one by one. If no match leads to a solution, the algorithm halts.

3.5.5 Selecting functional blocks

Given the functional block set F_B, this step searches for a subset that executes all payload statements. The goal is to select *exactly* one functional block for every IR statement and find dispatcher blocks to chain them together. BOPC builds the *delta graph* δG, described in Section 3.4.4.

Once the delta graph is generated, this step locates the *minimum* (in terms of total edge weight) *induced subgraph*, H_{k_0}, that contains the *complete* set of functional blocks to execute the SPL payload. If H_{k_0}, does not result in a solution, the algorithm tries the next minimum induced subgraph, H_{k_1}, until a solution is found or a limit is reached.

If the resulting delta graph does not lead to a solution, this step "shuffles" out-of-order payload statements, see Section 3.5.2, and builds a new delta graph. Note that the number of different permutations may be exponential. Therefore, our algorithm sets an *upper bound* P on the number of tried permutations.

Each permutation results in a different yet semantically equivalent SPL payload, so the CFG of the payload (called *Adjacency Matrix*, M_{Adj}) needs to be recalculated.

3.5.6 Discovering dispatcher blocks

The simulation phase takes the individual functional blocks (contained in the minimum induced subgraph H_{k_i}) and tries to find the appropriate dispatcher blocks to compose the

BOP gadgets. It returns a set of memory assignments for the corresponding dispatcher blocks, or an error indicating un-satisfiable constraints for the dispatchers.

BOPC is called to find a dispatcher path for *every* edge in the minimum induced subgraph. That is, we need to simulate every control flow transfer in the adjacency matrix, M_{Adj} of the SPL payload. However, dispatchers are *built* on the prior set of BOP gadgets and their impact on the binary's execution state so far, so BOP gadgets must be stitched with the respect to the program's current flow originating from the entry point.

Finding dispatcher blocks relies on concolic execution. Our algorithm utilizes functional block proximity as a metric for dispatcher path quality. However, it cannot predict which constraints will take exponential time to solve (in practice we set a timeout). Therefore concolic execution selects the K shortest dispatcher paths relative to the current BOP chain, and tries them in order until one produces a set of satisfiable constraints. It turns that this metric works well in practice even for small values of K (e.g., 8). This is similar to the *k-shortest path* [122] algorithm used for the delta graph.

When simulation starts it also initializes any SPL variables at the locations that are reserved during the variable mapping (Section 3.5.4). These addresses are marked as *immutable*, so any unintended modification raises an exception which stops this iteration.

In Table 3.3, we introduce the set of *Dereferenced Addresses*, D_M, which is the set of memory addresses whose contents are loaded into registers. Simulation cannot obtain the exact location of a symbolic address (e.g., `[rax + 4]`) until the block is executed and the register has a concrete value. Before simulation reaches a functional block, it concretizes any symbolic addresses from D_M and initializes the memory cell accordingly. If that memory cell has already been set, any initialization *prior* to the entry point cannot persist. That is, BOPC cannot leverage an AWP to initialize this memory cell and the iteration fails. If a memory cell has been used in the constraints, its concretization can make constraints unsatisfiable and the iteration may fail.

Simulation traverses the minimum induced subgraph, and *incrementally* extends the SPL state from one BOP gadget to the next, ensuring that newly added constraints remain satisfiable. When encountering a conditional statement (i.e., a functional block has two outgoing edges), BOPC *clones* the current state and continues building the trace for both paths independently, in the same way that a symbolic execution engine handles conditional statements. When a path reaches a functional block that was already visited, it gracefully terminates. At the end, we collect all those states and check whether the constraints of all these paths are satisfied or not. If so, we have a solution.

3.5.7 Synthesizing exploits

If the simulation module returns a solution, the final step is to encode the execution trace as a set of memory writes in the target binary. The constraint set C_w collected during simulation reveals a memory layout that leads to a flow across functional blocks according to the minimum induced subgraph. Concretizing the constraints for all participating conditional variables at the *end* of the simulation can result in incorrect solutions. Consider the following case:

```
a = input();
if (a > 10 && a < 20) {
        a = 0;
        /* target block */
}
```

The symbolic execution engine concretizes the symbolic variable assigned to a upon assignment. When execution reaches “target block”, a is 0, which is contradicts the precondition to reach the target block. Hence, BOPC needs to resolve the constraints *during* (i.e., *on the fly*), rather than at the end of the simulation.

Therefore, constraints are solved inline in the simulation. BOPC carefully monitors all variables and concretizes them at the “right” moment, just before they get overwritten. More specifically, memory locations that are accessed for first time, are assigned a symbolic variable. Whenever a memory write occurs, BOPC checks whether the initial symbolic variable still exists in the new symbolic expression. If not, BOPC concretizes it, adding the concretized value to the set of memory writes.

There are also some symbolic variables that do *not* participate in the constraints, but are used as pointers. These variables are concretized to point to a writable location to avoid segmentation faults outside of the simulation environment.

Finally, it is possible for registers or external symbolic variables (e.g., data from stdin, sockets or file descriptors) to be part of the constraints. BOPC executes a similar translation for the registers and any external input, as these are inputs to the program that are usually also controlled by the attacker.

3.6 Evaluation

To evaluate BOPC, we leverage a set of 10 applications with known memory corruption CVEs, listed in Table 3.4. These CVEs correspond to arbitrary memory writes [15, 16, 133], fulfilling our AWP primitive requirement. Table 3.4 contains the total number of all functional blocks for each application. Although there are many functional blocks, the difficulty of finding stitchable dispatcher blocks makes a significant fraction of them unusable.

Basic block abstraction is a time consuming process – especially for applications with large CFGs – but these results may be reused across iterations. Thus, as a performance optimization, BOPC caches the resulting abstractions of the Binary Frontend (Figure 3.5) to a file and loads them for each search, thus avoiding the startup overhead listed in Table 3.4.

To demonstrate the effectiveness of our algorithm, we chose a set of 13 representative SPL payloads [2] shown in Table 3.5. Our goal is to “map and run” each of these payloads on top each of the vulnerable applications. Table 3.6 shows the results of running each payload. BOPC successfully finds a mapping of memory writes to encode an SPL payload as a set of side effects executed on top of the applications for 105 out of 130 cases, approximately 81%. In each case, the memory writes are sufficient to reconstruct the payload execution by strictly following the CFG without violating a strict CFI policy or stack integrity.

[2] Results depend on the SPL payloads and the vulnerable applications. We chose the SPL payloads to showcase all SPL features, other payloads or combination of payloads are possible. We encourage the reader to play with the open-source prototype.

Table 3.4.: **Vulnerable applications**. The *Prim.* column indicates the primitive type (`AW` = Arbitrary Write, `FMS` = ForMat String). *Time* is the amount of time needed to generate the abstractions for every basic block. *Functional blocks* show the total number for each of the statements (*RegSet* = Register Assignments, *RegMod* = Register Modifications, *MemRd* = Memory Load, *MemWr* = Memory Store, *Call* = system/library calls, *Cond* = Conditional Jumps). Note that the number of call statements is small because we are targeting a predefined set of calls. Also note that MemRd statements are a subset of RegSet statements.

Vulnerable Application			**CFG**		**Time (m:s)**	**Total number of functional blocks**						
Program	**Vulnerability**	**Prim.**	**Nodes**	**Edges**		**RegSet**	**Reg-Mod**	**MemRd**	**MemWr**	**Call**	**Cond**	**Total**
ProFTPd	CVE-2006-5815 [123]	AW	27,087	49,862	10:08	40,143	387	1,592	199	77	3,029	45,427
nginx	CVE-2013-2028 [124]	AW	24,169	44,645	12:36	31,497	1,168	1,522	279	35	3375	37,876
sudo	CVE-2012-0809 [125]	FMS	3,399	6,267	01:14	5,162	26	157	18	45	307	5715
orzhttpd	BugtraqID 41956 [126]	FMS	1,354	2,163	00:27	2,317	9	39	8	11	89	2473
wuftdp	CVE-2000-0573 [127]	FMS	8,899	17,092	03:22	14,101	62	274	11	94	921	15,463
nullhttpd	CVE-2002-1496 [128]	AW	1,488	2,701	00:27	2,327	77	54	7	19	125	2,609
opensshd	CVE-2001-0144 [129]	AW	6,688	12,487	01:53	8,800	98	214	19	63	558	9,752
wireshark	CVE-2014-2299 [130]	AW	74,186	162,111	29:41	12,4053	639	1,736	193	100	4555	131276
apache	CVE-2006-3747 [131]	AW	18,790	34,205	10:22	33,615	212	490	66	127	1,768	36,278
smbclient	CVE-2009-1886 [132]	FMS	166,081	351,309	82:25	265,980	1,481	6,791	951	119	28,705	304,027

Table 3.5.: SPL payloads. Each payload consists of $|S|$ statements. Payloads that produce *flat* delta graphs (i.e., have no jump statements), are marked with ✓. *memwr* payload modifies program memory on the fly, thus preserving the Turing completeness of SPL (recall from Section 3.3 that AWP/ARP-based state modification is no longer allowed).

Payload	**Description**	$\|S\|$	**flat?**
regset4	Initialize 4 registers with arbitrary values	4	✓
regref4	Initialize 4 registers with pointers to arbitrary memory	8	✓
regset5	Initialize 5 registers with arbitrary values	5	✓
regref5	Initialize 5 registers with pointers to arbitrary memory	10	✓
regmod	Initialize a register with an arbitrary value and modify it	3	✓
memrd	Read from arbitrary memory	4	✓
memwr	Write to arbitrary memory	5	✓
print	Display a message to stdout using `write`	6	✓
execve	Spawn a shell through `execve`	6	✓
abloop	Perform an arbitrarily long bounded loop utilizing regmod	2	✗
infloop	Perform an infinite loop that sets a register in its body	2	✗
ifelse	An if-else condition based on a register comparison	7	✗
loop	Conditional loop with register modification	4	✗

Table 3.6 shows that applications with large CFGs result in higher success rates, as they encapsulate a “richer” set of BOP gadgets. Achieving truly infinite loops is hard in practice, as most of the loops in our experiments involve some loop counter that is modified in each iteration. This iterator serves as an index to dereference an array. By falsifying the exit condition through modifying loop variables (i.e., the loop becomes infinite), the program eventually terminates with a segmentation fault, as it tries to access memory outside of the current segment. Therefore, even though the loop would run forever, an external factor (segmentation fault) causes it to stop. BOPC aims to address this issue by simulating the same loop multiple times. However, finding a truly infinite loop requires BOPC to simulate it an infinite number of times, which is infeasible. For some cases, we managed to verify that the accessed memory inside the loop is bounded and therefore the solution truly is an infinite loop. Otherwise, the loop is *arbitrarily bounded* with the upper bound set by an external factor.

For some payloads, BOPC was unable to find an exploit trace. This is is either due to imprecision of our algorithm, or because no solution exists for the written SPL payload. We can alleviate the first failure by increasing the upper bounds and the timeouts in our configuration. Doing so, makes BOPC search more exhaustively at the cost of search time.

Table 3.6.: Feasibility of executing various SPL payloads for each of the vulnerable applications. An ✓ means that the SPL payload was successfully executed on the target binary while a ✗ indicates a failure, with the subscript denoting the type of failure (✗$_1$ = Not enough candidate blocks, ✗$_2$ = No valid register/variable mappings, ✗$_3$ = No valid paths between functional blocks and ✗$_4$ = Un-satisfiable constraints or solver timeout). Note that in the first two cases (✗$_1$ and ✗$_2$), we know that there is *no* solution while, in the last two (✗$_3$ and ✗$_4$), a solution might exists, but BOPC cannot find it, either due to over-approximation or timeouts. The numbers next to the ✓ in *abloop*, *infloop*, and *loop* columns indicate the maximum number of iterations. The number next to the *print* column indicates the number of character successfully printed to the stdout.

Program	**SPL payload**												
	regset4	*regref4*	*regset5*	*regref5*	*regmod*	*memrd*	*memwr*	*print*	*execve*	*abloop*	*infloop*	*ifelse*	*loop*
ProFTPd	✓	✓	✓	✓	✓	✓	✓	✓ 32	✗$_1$	✓ 128+	✓ ∞	✓	✓ 3
nginx	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✓	✓ 128+	✓ ∞	✓	✓ 128
sudo	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✓ 128+	✗$_4$	✗$_4$
orzhttpd	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✗$_1$	✗$_4$	✓ 128+	✗$_4$	✗$_3$
wuftdp	✓	✓	✓	✓	✓	✓	✓	✓	✗$_1$	✓ 128+	✓ 128+	✗$_4$	✗$_3$
nullhttpd	✓	✓	✓	✓	✓	✓	✗$_3$	✗$_3$	✓	✓ 30	✓ ∞	✗$_4$	✗$_3$
opensshd	✓	✓	✓	✓	✓	✓	✗$_4$	✗$_4$	✗$_4$	✓ 512	✓ 128+	✓	✓ 99
wireshark	✓	✓	✓	✓	✓	✓	✓	✓ 4	✗$_1$	✓ 128+	✓ 7	✓	✓ 8
apache	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✗$_4$	✓ ∞	✓ 128+	✓	✗$_4$
smbclient	✓	✓	✓	✓	✓	✓	✓	✓ 1	✗$_1$	✓ 1057	✓ 128+	✓	✓ 256

The failure to find a solution exposes the *limitations* of the vulnerable application. This type of failure is due to the "structure" of the application's CFG, which prevents BOPC from finding a trace for an SPL payload. Hence, a solution may not exist due to one the following:

1. There are not enough candidate blocks or functional blocks.
2. There are no valid register / variable mappings.
3. There are no valid paths between functional blocks.
4. The constraints between blocks are unsatisfiable or symbolic execution raised a timeout.

For instance, if an application (e.g., ProFTPd) never invokes `execve` then there are no candidate blocks for `execve` SPL satements. Thus, we can infer from the *execve* column in Table 3.6 that all applications with a ✗$_1$ never invoke `execve`.

In Section 3.3 we mention that the determination of the entry point is part of the vulnerability discovery process. Therefore, BOPC assumes that the entry point is given. Without having access to actual exploits (or crashes), the locations of entry points are ambiguous. Hence, we have selected arbitrary locations as the entry points. This allows BOPC to find payloads for the evaluation without having access to *concrete* exploits. In practice, BOPC would leverage the given entry points as starting points. We demonstrate several test cases where the entry points are precisely at the start of functions, deep in the Call Graph, to show the power of our approach. Orthogonally, we allow for vulnerabilities to exist in the middle of a function. In such situations, BOPC would set our entry point to the location *after* the return of the function.

The lack of the exact entry point complicates the verification of our solutions. We leverage a debugger to "simulate" the AWP and modify the memory on the fly, as we reach the given entry point. We ensure as we step through our trace that we maintain the properties of the SPL payload expressed. That is, blocks between the statements are non-clobbering in terms of register allocation and memory assignment.

3.7 Case Study: nginx

We utilize a version of the nginx web server with a known memory corruption vulnerability [124] that has been exploited in the wild to further study BOPC. When an HTTP header contains the "Transfer-Encoding: chunked" attribute, nginx fails to properly bounds check the received packet chunks, resulting in stack buffer overflow. This buffer overflow [15] results in an arbitrary memory write, fulfilling the AWP requirement. For our case study we select three of the most interesting payloads: spawning a shell, an infinite loop, and a conditional branch. Table 3.7 shows metrics collected during the BOPC execution for these cases.

Table 3.7.: Performance metrics (run on Ubuntu 64-bit with an i7 processor) for BOPC on nginx. *Time* = time to synthesize exploit, $|C_B|$ = # candidate blocks, *Mappings* = # concrete register and variable mappings, $|\delta G|$ = # delta graphs created, $|H_k|$ = # of induced subgraphs tried.

Payload	**Time**	$\lvert C_B \rvert$	**Mappings**	$\lvert \delta G \rvert$	$\lvert H_k \rvert$
execve	0m:55s	10,407	142,355	1	1
infloop	4m:45s	9,909	14	1	1
ifelse	1m:47s	10,782	182	4	2

3.7.1 Spawning a shell

Function `ngx_execute_proc` is invoked through a function pointer, with the second argument (passed to `rsi`, according to x64 calling convention), being a **void** pointer that is interpreted as a `struct` to initialize all arguments of `execve`:

```
mov   rbx, rsi
mov   rdx, QWORD PTR [rsi+0x18]
mov   rsi, QWORD PTR [rsi+0x10]
mov   rdi, QWORD PTR [rbx]
call  0x402500 <execve@plt>
```

BOPC leverages this function to successfully synthesize the `execve` payload (shown on the right side of Table 3.1) and generate a PoC exploit in less than a minute as shown in Table 3.7.

Assuming that `rsi` points to some writable address x, BOPC produces the following $(address, value, size)$ tuples: $(\$y, \$x, 8)$, $(\$y + 8h, 0, 8)$, $(\$x, /bin/sh, 8)$, $(\$x + 10h, \$y, 8)$, $(\$x + 18h, 0, 8)$, were $\$y$ is a concrete writable addresses set by BOPC.

3.7.2 Infinite loop

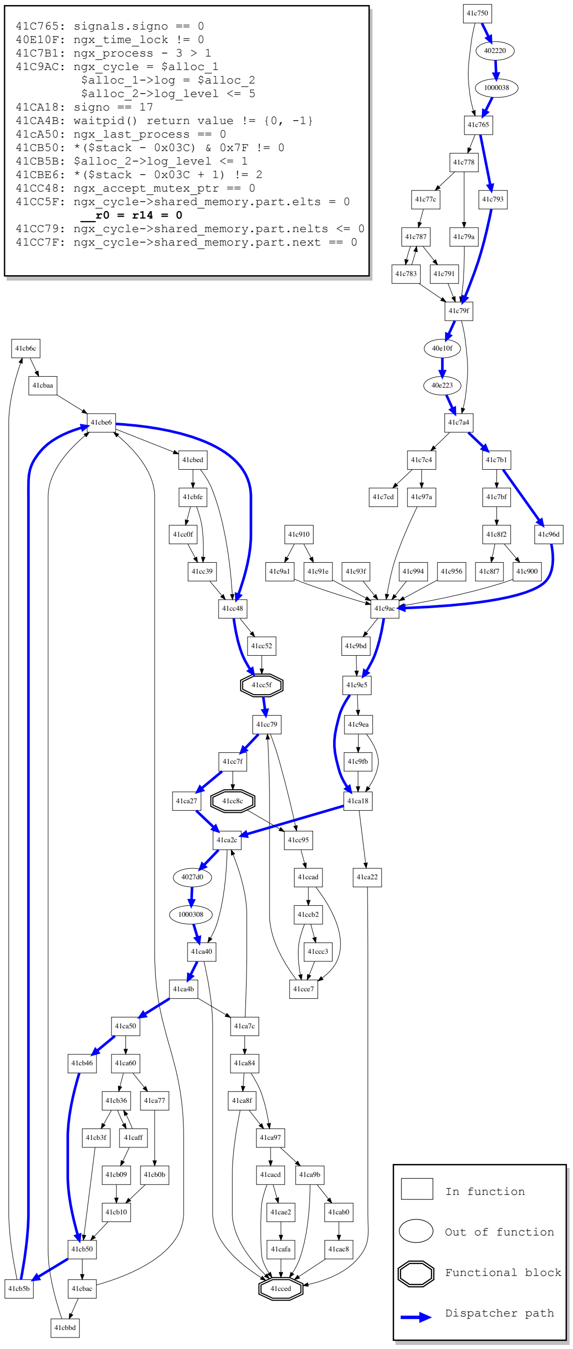
Here we present a payload that generates a trace that executes an infinite loop. The *infloop* payload is a simple infinite loop that consists of only two statements:

```
void payload() {
  LOOP:
    __r1 = 0;
    goto LOOP;
}
```

We set the entry point at the beginning of `ngx_signal_handler` function which is a signal handler that is invoked through a function pointer. Hence, this point is reachable through control-flow hijacking. The solution synthesized by BOPC is shown in Figure 3.6. The box on the top-left corner demonstrates how the memory is initialized to satisfy the constraints.

Virtual register `__r0` was mapped to hardware register `r14`, so `ngx_signal_handler` contains three candidate blocks, marked as octagons. Exactly one of them is selected to be the functional block while the others are avoided by the dispatcher blocks. The dispatcher finds a path from the entry point to the first functional block, and then finds a loop to return back to the same functional block (highlighted with blue arrows). Note that the size of the dispatcher block exceeds 20 basic blocks while the functional block consists of a single basic block.

The oval nodes in Figure 3.6 indicate basic blocks that are outside of the current function. At basic block `0x41C79F`, function `ngx_time_sigsafe_update` is invoked. Due to



```
41C765: signals.signo == 0
40E10F: ngx_time_lock != 0
41C7B1: ngx_process - 3 > 1
41C9AC: ngx_cycle = $alloc_1
         $alloc_1->log = $alloc_2
         $alloc_2->log_level <= 5
41CA18: signo == 17
41CA4B: waitpid() return value != {0, -1}
41cA50: ngx_last_process == 0
41CB50: *($stack - 0x03C) & 0x7F != 0
41CB5B: $alloc_2->log_level <= 1
41CBE6: *($stack - 0x03C + 1) != 2
41CC48: ngx_accept_mutex_ptr == 0
41CC5F: ngx_cycle->shared_memory.part.elts = 0
         __r0 = r14 = 0
41CC79: ngx_cycle->shared_memory.part.nelts <= 0
41CC7F: ngx_cycle->shared_memory.part.next == 0
```

Figure 3.6.: CFG of nginx's `ngx_signal_handler` and payload for an infinite loop (blue arrow dispatcher blocks, octagons functional blocks) with the entry point at the function start. The top box shows the memory layout initialization for this loop. This graph was created by BOPC.

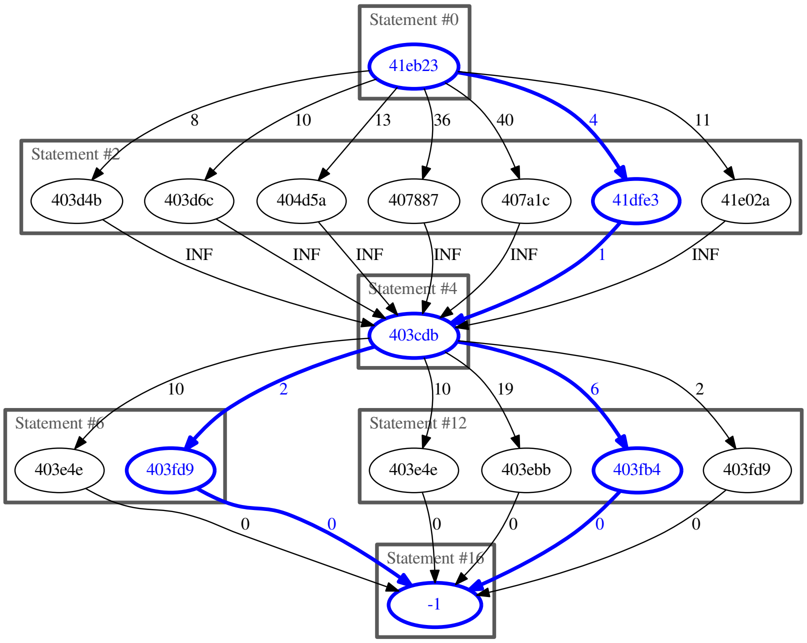


Figure 3.7.: A delta graph instance for an *ifelse* payload for nginx. The first node is the *entry point*. Blue nodes and edges form the *minimum induced subgraph*, H_k. Statement $\#4$ is a conditional, execution branches into two statements. Note that BOPC created this graph.

the shortest path heuristic, BOPC, tries to execute as few basic blocks as possible from this function. In order to do so BOPC sets `ngx_time_lock` a non-zero value, thus causing this function to return quickly. BOPC successfully synthesizes this payload in less than 5 minutes.

3.7.3 Conditional statements

This case study shows an SPL if-else condition that implements a logical NOT. That is, if register `__r0` is zero, the payload sets `__r1` to one, otherwise `__r1` becomes zero. The execution trace starts at the beginning of `ngx_cache_manager_process_cycle`. This function is called through a function pointer. A part of the CFG starting from this function is shown in Appendix 7.6. After trying 4 mappings, `__r0` and `__r1` map to `rsi` and `r15` respectively. The resulting delta graph is the shown in Figure 3.7.

As we mentioned in Section 3.5.6, when BOPC encounters a functional block for a conditional statement, it *clones* the current state of the symbolic execution and the two clones independently continue the execution. The constraints up to the conditional jump are the following:

```
0x41eb23 : $rdi = ngx_cycle_t* cycle
0x40f709 : *(ngx_event_flags + 1) == 0x2
0x41dfe3 : __r0 = rsi = 0x0
0x403cdb : $r15 = 0x1
           ngx_module_t ngx_core_module.index = 0
           $alloca_1 = *cycle
           ngx_core_conf_t* conf_ctx =
               *$alloca_1 + ngx_core_module.index * 8
0x403d06 : test rsi, rsi (__r0 != 0)
0x403d09 : jne  0x403d1b <ngx_set_environment+64>
```

If the condition is false and the jump is *not* taken, the following constraints are also added to the state.

```
0x403d0b : conf_ctx->environment != 0
0x403fd9 : __r1 = *($stack - 0x178) = 1;
```

When the condition is true, the execution trace will follow the “taken” branch of the trace. In this case the shortest path to the next functional block is $403d1b \rightarrow 403d3d \rightarrow 403d4b \rightarrow 403d54 \rightarrow 403d5a \rightarrow 403fb4$ with a total length 6. Unfortunately, this cannot be used as a dispatcher block, due to an exception that is raised at $403d4b$. The register `rsi`, is 1 and therefore when we attempt to execute the following instruction: `cmp BYTE PTR [rsi], 54h`, we essentially try to dereference address 1. BOPC is aware of this exception, so it discards the current path and tries with the second shortest path. The second shortest path has length 7 and avoids the problematic block: $403d1b \rightarrow 403d8b \rightarrow 4050ba \rightarrow 40511c \rightarrow 40513a \rightarrow 403d9c \rightarrow 403da5 \rightarrow 403fb4$. This results in a new set of constraints as shown below:

```
0x403d1b : conf_ctx->env.elts = &elt (ngx_array_t*)
           conf_ctx->env.nelts == 0
0x4050ba : conf_ctx->env.nelts != $alloca_2->env.nalloc
0x40511c : conf_ctx->env.nelts += 1
0x40513a : $ret = conf_ctx->env.elts +
                  conf_ctx->env.nelts*conf_ctx->env.size
0x403d9c : $ret != 0
0x403da5 : conf_ctx->env.nelts != 0
0x403fb4 : __r1 = r15 = 0
```

3.8 Discussion and Future Work

Our prototype demonstrates the feasibility and scalability of automatic construction of BOP chains through a high level language. However, we note some potential optimizations that we will consider for future versions of BOPC.

BOPC is limited by the *granularity* of basic blocks. That is, a combination of basic blocks could potentially lead to the execution of a desired SPL statement, while individual blocks might not. Take for instance an instruction that sets a virtual register to 1. Assume that a basic block initializes `rcx` to 0, while the following block increments it by 1; a pattern commonly encountered in loops. Although there is no functional block that directly sets `rcx` to 1, the combination of the previous two has the desired effect. BOPC can be expanded to address this issue if the basic blocks are coalesced into larger blocks that result in a new CFG.

BOPC sets several upper bounds defined by user inputs. These configurable bounds include the upper limit of (i) SPL payload permutations (P), (ii) length of continuous blocks (L), (iii) of minimum induced subgraphs extracted from the delta graph (N), and (iv) dispatcher paths between a pair of functional blocks (K). These upper bounds along with the timeout for symbolic execution, reduce the search space, but prune some potentially valid solutions. The evaluation of higher limits may result in alternate or more solutions being found by BOPC.

3.9 Conclusion

Despite the deployment of strong control-flow hijack defenses such as CFI or shadow stacks, data-only code reuse attacks remain possible. So far, configuring these attacks relies on complex manual analysis to satisfy restrictive constraints for execution paths.

Our BOPC mechanism automates the analysis of the remaining attack surface and synthesis of exploit payloads. To abstract complexity from target programs and architectures, the payload is expressed in a high-level language. Our novel code reuse technique, *Block Oriented Programming*, maps statements of the payload to functional basic blocks. Functional blocks are stitched together through dispatcher blocks that satisfy the program CFG and avoid clobbering functional blocks. To find a solution for this NP-hard problem, we develop heuristics to prune the search space and to evaluate the most probable paths first.

The evaluation demonstrates that the majority of 13 payloads, ranging from typical exploit payloads to loops and conditionals are successfully mapped 81% of the time across 10 programs. Upon acceptance, we will release the source code of our proof of concept prototype along with all of our evaluation results. The prototype is available at `https://github.com/HexHive/BOPC`.

4 X-CAP: ASSESSING EXPLOITATION CAPABILITIES

In Chapter 3 we introduced a novel technique, called *Block Oriented Programming* (BOP), to automate data-only attacks. The main intuition behind Block Oriented Programming is, given an exploit payload, to find a sequence "gadgets" that perform useful computations (called *functional gadgets*), and stitch them together through a sequence of *dispatcher gadgets*. The purpose of a dispatcher gadget is twofold: *First*, it assures the smooth transition between two functional gadgets, without clobbering the the execution state (or context) that functional gadgets build. *Second*, it ensures that program's execution flow abides with Control Flow Graph (CFG) and therefore never violates Control Flow Integrity (CFI).

However, the problem of stitching functional gadgets is NP-hard as it reduces from K-Clique problem (see Section 7.4 for a detailed proof). Furthermore, it also involves the use of symbolic execution and constraint solving, two problems that reduce from 3-SAT [134], the original NP-complete problem. Hence, despite the extensive effort that our framework, BOPC [17], puts to stitch all functional gadgets together, there is no guarantee that such a solution will exist, as shown in Section 3.6.

A closer look at the cases were BOPC fails, reveals an interesting problem: Inferring the *root cause* of the failures. BOPC has inherent limitations as it deals with NP-hard problems. Therefore, it may not be capable of finding a solution all the times. But what if a solution does not exist at all? In this chapter we aim to formulate this problem and determine under which circumstances it is infeasible to stitch two functional gadgets together. Our analysis results in three possible outcomes:

- It is possible to stitch two functional gadgets together, as BOPC has found a solution (*proven connectivity*).

- It is impossible to stitch two functional gadgets together, because gadgets are either too far apart or they have unsatisfiable constraints (*proven disconnectivity*).
- We have good indications that it may not be possible to stitch two functional gadgets together. BOPC did not find a solution because either a timeout was raised during concolic execution, or a potential solution pruned from the search space. (*potential dis-connectivity*).

Although we can reduce the probability of falling in the last case by repeating the experiment with longer timeouts and a more extensive search, there is no guarantee that we can avoid it, as the execution time and the search space can be exponential. Nevertheless, BOPC has the ability to distinguish between the last two cases.

Therefore, we can leverage the second outcome (proven disconnectivity) to solve the inverse problem: Finding which functional gadgets are *impossible* to stitch together. This is an interesting outcome, because if we know that it is infeasible to stitch two functional gadgets together, we can infer that it is not possible to be part of the same payload. That is, we know what payloads an adversary, is *not* capable of executing on a vulnerable application.

We can formalize the previous statement and assess the *exploitation capabilities* on a vulnerable application. Our tool, X-Cap, leverages BOPC to find functional gadgets that impossible to stitch together and functional gadgets that is feasible to stitch together. X-Cap encodes this information in a directed graph, called *capability graph*. In this graph each node represents functional gadget and each edge the potential connectivity between two gadgets. An interesting property of this graph is that it constitutes from several, disconnected components, called *islands* and they essentially represent gadget reachability.

By analyzing the capability graph we can infer that if two functional gadgets belong to different computation islands then it is impossible to coexist in the same exploit payload. However when two functional gadgets belong to the same computation island it does not necessary mean that we can always stitch them together. For instance consider three functional gadgets A, B and C, as shown in Figure 4.1, that reside on the same computational island. Although it is possible to stitch A with B, B with C and A with C together, it is

```
/* declare a guard variable */
var_a = input();

/* code that executes gadget A */
gadget_A();


if (var_a == 0) {
        /* code that executes gadget B */
        gadget_B();
}

if (var_a == 1) {
        /* code that executes gadget C */
        gadget_C();
}
```

Figure 4.1.: Code snippet that shows computation island disconnectivity. Here, we have three functional gadgets A, B and C. Although it is possible to stitch A with B, A with C and B with C together (so all of A, B and C, belong to the same computation island), it is impossible to stitch all of them together as this requires var_a to be 0 and 1 at the same time. This is because the path constrains contradict and therefore become unsatisfiable.

impossible to stitch A, B and C all together. This is because their imposed constrains that are built along the path contradict: The prerequisite to stitch A and B requires specific a memory address to be zero, while the prerequisite to stitch B and C is the same memory address to be nonzero.

Therefore, the computation islands give us *upper bounds*. That is, they indicate the largest set of functional gadgets that can be on the same payload, in the best case scenario that all constrains are satisfiable. This allows X-Cap to infer properties regarding the *composition* of the exploit payloads that can be executed under a vulnerable application.

In Chapter 3, we described how BOPC indicates whether the Residual Attack Surface is non-zero or not, by finding *an* exploit payload that can be executed under binaries hardened with advanced mitigations, such as CFI and shadow stacks. Here, X-Cap identifies sets of exploit payloads that can successfully be executed on top of a (given) vulnerable application. We refer to this term as *application's capability* and essentially encapsulates all "properties" that an exploit payload should carry to be successful executed.

Large applications likely contain vulnerabilities so our tool, X-Cap, allows for an assessment of the exploitability of a vulnerable application. X-Cap follows a similar approach with BOPC and therefore it reuses a large portion from it. It finds all potential (individual) SPL statements that the vulnerable application is capable of executing and builds the proximity graph, which provides strong indications on which SPL statements together can be stitched together. X-Cap is an ongoing work.

5 MALWASH: WASHING MALWARE TO EVADE DYNAMIC ANALYSIS

Hiding malware processes from fingerprinting is challenging. Current techniques like metamorphic algorithms and diversity generate different instances of a program, protecting it against static detection. Unfortunately, all existing techniques are prone to detection through behavioral analysis – a runtime analysis that records behavior (e.g., through system call invocations), and can detect executing diversified programs like malware.

We present malWASH, a dynamic diversification engine that executes an arbitrary program without being detected by dynamic analysis tools. Target programs are chopped into small components that are then executed in the context of other processes, hiding the behavior of the original program in a stream of benign behavior of a large number of processes. A scheduler connects these components and transfers state between the different processes. The execution of the benign processes is not impacted. Furthermore, malWASH ensures that the executing program remains persistent, complicating the removal process.

5.1 Introduction

Malware (and fighting malware) is an important aspect of computer security. Malware by itself does not exploit security vulnerabilities but is the payload that is executed post-exploitation. Consequently, malware is only successful if it is stealthy and remains undetected. Sophisticated, undetectable malware is therefore a required asset for attackers. Anti Virus systems (AV) are based on signature detection and static analysis. Although this method has limitations, it is well-proven, reliable, and accurate. The AV identifies malware by looking for known patterns or characteristics. Due to its simplicity and accuracy, signature-based detection remains widely used.

Malware authors bypass signature-based detection by using metamorphic [26] algorithms and diversity. These techniques generate instances of the same binary that have different signatures, while maintaining the functionality of the binary. Defenders quickly realized that all generated instances have the same functionality, and started to identify the behavior of the malware instead of the signature [22]. Dynamic analysis executes the malware to reveal its behavior. This method is simple but effective, e.g., a typical keylogger repeatedly performs a sequence of specific system calls. No matter how obfuscated the binary is, these system calls are repeated in the same order, making the keylogger easily detectable.

A simple technique to bypass behavior based detection would be to insert bogus system calls (i.e., system calls that do not affect the original execution) between real ones. An analysis can likely filter out bogus system calls, thereby mitigating this naive technique. We propose a sophisticated, novel mechanism to hide malware from behavior-based analysis. Rather than executing the program in a single process, we automatically distribute the program across a set of pre-existing, benign processes. Our approach is based on a simple observation: although we cannot modify the executing system calls and their order of execution in a binary, we can hide them within the stream of system calls that are performed on the *entire* system.

To spread our system calls across the stream of calls for the *entire* system we propose injecting our system calls into a set of existing processes on the system. To do this, the original binary is chopped into small chunks. Each individual chunk only contains limited functionality and therefore executes few system calls. These small chunks and an "*emulator*" are then injected into multiple running processes and blend into the stream of executed system calls. Each emulator then selects the individual chunks to run, captures state, and coordinates with the other emulators who continues execution.

Detection tools that observe behavior based on a per-process analysis no longer see the complete sequence of system calls that the program executes. Each injected system call is hidden in a set of benign system calls and the program functionality is spread across a set of benign processes, executing benign code (in addition to the injected one). Tracking the system calls of all applications globally and trying to look for malicious patterns is a strictly harder problem, as system calls from the injected binary are spread out in the stream of system calls for the entire system. Consequently, methods like [29] which search for short sequences of malicious system calls fail.

Prior obfuscation techniques such as [135, 136] guarantee that the actual computation remains the same, which is a required, fundamental property that enabled behavioral analysis. malWASH guarantees equal functionality, while bypassing behavioral analysis. The design of our "malware" engine allows chopping and executing arbitrary programs. To keep our Windows-based prototype implementation general but simple, we constrain the execution environment, and assume that the binary has some specific properties (defined in Section 5.4). We evaluate our malWASH prototype implementation with samples from different malware categories and show that our implementation successfully chops and executes the programs.

Beyond stealthiness, malWASH offers another interesting property: *resilience*. The malware is distributed as it is injected into multiple benign processes and executes as part of them. Therefore, killing a single process does not stop the execution of the malware as it can reinstantiate itself from any remaining emulator. The only way to stop malWASH is to kill *all* infected processes at the same time, before any process reinfects a new process.

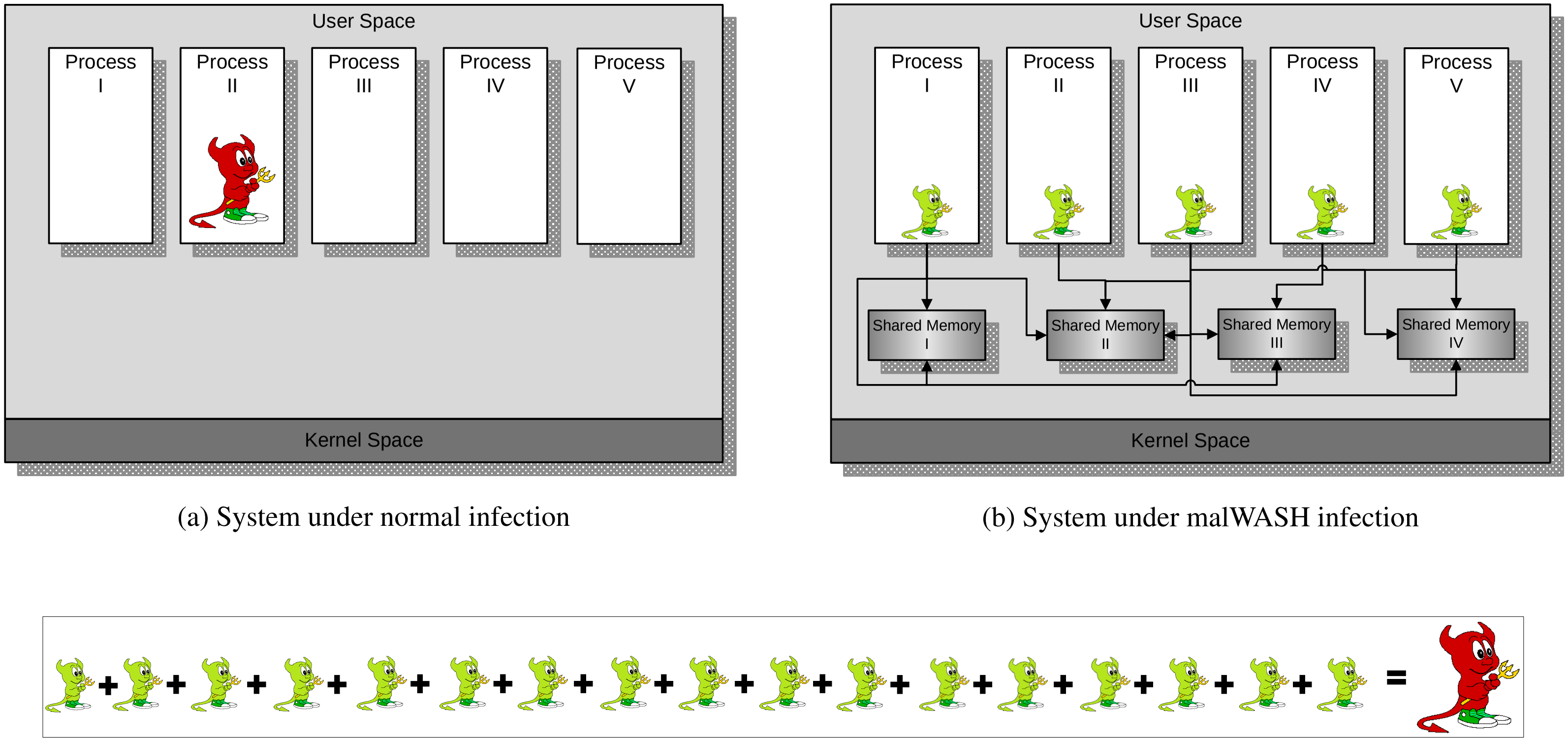
The contributions of malWASH are:

- Design of an execution engine that thwarts behavioral and dynamic analysis.
- Creation of fully persistent malware that continues executing as long as at least one emulator remains.

Furthermore, the design of malWASH, has some very interesting properties. First, even if malWASH is detected, the actual binary remains obfuscated in a plethora of processes, complicating reverse engineering. An analyst would first have to correctly reassemble the binary. Second, all of the existing obfuscation and diversity techniques can be used with malWASH.

5.2 Background and Related Work

Over the last decade, many techniques have been proposed to enable obfuscation and diversity, with the goal of hiding malware from AV systems. One of the oldest methods



(a) System under normal infection

(b) System under malWASH infection

(c) Conceptually, all the small (and benign) injected parts are equal with the original malware

Figure 5.1.: A comparison between normal infection and malWASH

to detect whether a given binary is malicious or not is to use static analysis detection [23, 25]. Anti-disassembly mechanisms [36, 136, 137] allow malware authors to bypass static analysis and companies to protect their IP against, e.g., illegal distribution. Although powerful, anti-disassembly techniques are infamous; benign programs have no reason to obfuscate their code as obfuscation may impact performance, stability, and the ability to reproduce crashes. Even though analysis of binaries protected by anti-disassembly is hard, it is straight-forward to check whether such protections were applied, e.g., detecting an encrypted PE header [24]. An AV can exploit this fact and flag a binary as malicious without trying to analyze it, as using such obfuscation is a strong indication that the binary is actually malicious.

Furthermore, these mechanisms have to eventually *reveal* their payloads and execute it. Techniques like dynamic analysis and sandboxes, analyze the malware and compare the behavior against well-known patterns. Anti-debugging techniques [39, 135] along with VM-detection [38] are used to change a program's behavior when a sandbox or a debugger is detected. All these methods share that the actual execution of the malware, when not being debugged, remains constant (this is a guaranteed property). Consequently, observing the behavior of the executing malware always yields the same observation (e.g., same system calls in the same order).

Improved obfuscation mechanisms were proposed, notably using Return Oriented Programming (ROP) to hide a malware within a benign program [138, 139]. Although effective, a ROP-style execution can easily be detected [106, 107, 108, 109, 110]. Another interesting obfuscation technique is movfuscator [140], which compiles a program using only `mov` instructions. This makes analysis extremely hard, but detecting that movfuscator is applied to a given binary is trivial. Any use of movfuscator is an indication that a binary is malicious, even if there is no information on what the binary does.

The concept behind the previous approaches, was to *hide* a malicious payload within a program. Another approach is to "get rid" of the malicious payload, by forcing another program to execute it for you. Metasploit's meterpreter [141] uses DLL and Reflective DLL [142] injection, to inject a malicious payload into another process's address space.

The common property of all the aforementioned protections is that any malicious action happens within the context of a single process. Here is where dynamic analysis [27] takes place. This is a powerful method that tries to classify a program or a process as malicious by observing its behavior (e.g., system calls, involved files, or network connections). Using dynamic analysis, it is possible to detect new, unknown malware just by matching the behavior of the binary.

Many dynamic analysis methods have been proposed to detect malware. Methods based on execution tracing [22, 28, 29, 30, 31, 32], inspect executing traces, looking for malicious patterns of system calls. However, when a binary runs under malWASH, is significantly complicated due to the distributed nature of our approach: (i) the execution trace of a process, contains only a small and out-of-order subset of system calls, and (ii) any sequence of system calls of the original binary is distributed among multiple processes, because each quantum given to malWASH, contains only a few system calls (e.g., 1 or 2).

The most recent malware detection methods use machine learning techniques to classify a binary as malicious or not [33, 34, 35, 143, 144]. However, these methods all assume that the malware runs in a single process and that only malicious system calls are executed by a process.

Even though the original binary is well hidden and protected, defenders could try to detect the malWASH emulator itself and not the binary it emulates. However, the idea of malWASH can also be used to protect malWASH itself. As we show in Section 5.5.3, the use of sub-emulators (small emulators that emulate the original emulator) along with other hardening methods in emulator, makes detection challenging.

5.3 Design

The design of malWASH follows a simple concept: breaking a program into small pieces and hiding these pieces in benign processes (see Figure 5.1). Conceptually, malWASH works as an emulator that (i) executes individual instructions of the program and (ii) coordinates with the other active emulators to create a correct flow of execution.

Behavioral malware detection is carried out per process (or per thread). After analysis, an *individual* process can be flagged as malicious. We believe that scaling behavioral analysis to a group of processes or threads is hard due to the exponential explosion of possible combinations of system calls across processes. malWASH introduces an emulator that allows the execution of a target program in a set of host processes. In the most stealthy mode of malWASH, a host process executes a single instruction of the emulated program per time slice. Detecting this one instruction within the millions of instructions that get executed by the process is highly unlikely.

malWASH takes as input a binary file and produces a C++ source file that embeds all the required parts of the binary along with all malWASH components. Using a source file as output enables further binary obfuscation processes (e.g., metamorphism). This means that all the existing protection methods against static analysis and signature detection work on top of malWASH.

malWASH operates in two phases. In the first phase, the original program binary is "chopped" into hundreds of small pieces and all the required information is extracted from the binary (segments, loaded libraries, relocations, global data and thread information). All these components (including those from malWASH) are encoded as character arrays and packed into a single C++ source file.

Chopping the binary into components is challenging as control-flow transfer instructions (e.g., `jcc`, `jmp`, `call`, and `ret`) may transfer control of the execution to a point that is not in the current address space. Therefore, the initial chopping is done at the basic block level. This way we know that only the last instruction transfers control to other locations. Using an emulator lookup function, we can replace the original instruction with a set of instructions that recover control-flow (possibly signaling another process to continue). Once we finish chopping on basic blocks, we can further chop the basic blocks themselves into new, smaller blocks, or start coalescing basic blocks to larger blocks.

Splitting opens a trade-off between *efficiency* and *stealthiness*. Using smaller blocks, malware signatures disappear and dynamic analysis detection tools fail to observe malicious behavior in the block. On the other hand, because there is a lot of overhead to transition between executed blocks (capturing state, selecting the next block to execute, and scheduling which process should execute the next block), fewer blocks will lead to less overhead from the emulator.

Once the source file is compiled, the program is ready to execute. The second phase of malWASH takes place when it starts execution. The first component is the loader, which looks for a set of “suitable” processes. The amount of emulators used is flexible and user-configurable. A good candidate is, e.g., the Google Chrome browser as it spawns many communicating processes that are perfect candidates for injection. A process is suitable when it allows another process to inject and execute code into its address space. Obviously, these instances need to cooperate, so a stealthy, reliable communication channel is needed. For this reason the loader also initializes a small set of shared memory regions for use by all the malWASH emulators. These shared regions contain data segments, stack, heap and all metadata that emulators are needed in order to cooperate and execute the blocks. Instead of shared memory, other communication mechanisms can be used such as pipes, files, network ports, or even covert channels.

Using shared memory regions has several advantages over process messages: (i) message may get lost and (ii) someone may observe messages between processes that are irrelevant to each other. If the emulator from process *A* communicates with the emulator from process *B*, it writes to the shared region that emulator of process *B* is waiting to read. Someone may still observe that there are new shared regions between processes, but as we show in Section 5.5.3, this information is of limited use.

Emulation of the malware begins after the loader terminates. Control is transferred to the first emulator (there is no central scheduling emulator) which executes its basic block, and then transfers control to the next emulator. At any time, exactly one emulator runs a piece of the original program (except for multi-threading programs where multiple emulators can execute different blocks of the program as long these blocks belong to different threads). Semaphores and Mutexes synchronize the emulators and ensure that no more than two processes will execute blocks from the same thread at each time.

When an emulator successfully takes the semaphore, it executes the next block of the malware. Before executing the next block, a context switch is performed and all memory accesses and imported function addresses are properly relocated. During the execution, current instructions within a block are executed transparently, without knowing of the emulation. After the block is executed, a context switch is performed, saving the current state of the program in the shared region and the emulators will coordinate to find which one will execute the next block. Note that different scheduling policies can be implemented to select which emulator executes the next block, we use a simple race. This distribution of emulators results in an address space independent execution.

When a process that contains an emulator terminates, the other emulators can continue the execution and the malware will continue to execute. The other emulators can detect the missing component and invoke the loader to reinitialize the missing emulator in a new process, keeping the total number of emulators constant. This means that as long as there is at least one emulator running it can recover from killed instances. Removing or stopping the malware requires that all emulators must be killed at the same time. The emulators run exclusively in memory, making it harder to detect as there are no persistent files.

5.4 Implementation

malWASH takes a binary program and distributes its execution across a set of benign processes, coordinating the global state of the program and the scheduling between the individual components. In the most fine-grained configuration, each instruction of the target program is a different entity. The Windows-based implementation of malWASH draws ideas from several areas: binary analysis to chop the program into individual components, binary translation to manipulate the control execution of each block, to coordinate between individual blocks and to orchestrate scheduling, and snapshotting to capture and synchronize program state across the different processes.

Our malWASH prototype implementation (available at `https://github.com/HexHive/malWASH`) consists of an offline and an online component. The offline compo-

Table 5.1.: List of supported properties by design and implemented in the current prototype.

Property	Prototype Implementation	Design
Obfuscated	No	Depends
Self Modifying	No	Yes
Polymorphic / Metamorphic	No	Yes
Packed	No	Yes
Anti disassembly	No	Yes
Anti debugging	(Yes)	(Yes)
Non PIE	Depends	Yes
Use Heap	Yes	Yes
Multi Threading	Yes	Yes
W+X sections	No	Yes
Non x86	No	Yes
Statically linked	Depends	Depends

nent runs the binary analysis, chops the program into individual components, and prepares the emulator. The online component includes the loader that injects components into different processes and the emulator which orchestrates and coordinates the execution of the program among all the different host processes.

By extracting the components offline, we can fall back on existing tools for the underlying binary analysis and, more importantly, our emulator does not require disassembly functionality. To keep the implementation prototype simple, we have restricted the (implemented) functionality of the emulator. Our emulator supports the full x86 instruction set (with a special focus of the control transfer instructions). Anti debugging features of the original binary can be mitigated by our translation and analysis process. The current implementation does not support x86-64 code and obfuscated or any form of self-modifying code (a design and engineering decision as otherwise the emulator would require its own binary analysis framework and disassembly functionality, vastly increasing the size of the emulator). Table 5.1 highlights the design trade-offs.

5.4.1 Phase 1: Chopping the binary

malWASH uses an IDA pro plugin to “chop” the binary. If IDA fails to analyse the binary, our tool will fail as well. Our plugin uses a Depth First Search (DFS) to disassemble the program from its entry point. This disassembly phase recursively follows control-flow transfer instructions and thereby recovers the Control-Flow Graph (CFG) of the binary, assigning a Block IDentifier (BID) to each basic block. These initial basic blocks can further be chopped into smaller pieces, depending on the configuration:

BBS (Basic Block Split) mode: the basic blocks are used as is.

BAST (Below AV Signature Threshold) mode: basic blocks are chopped so that each block is below a configurable threshold (we used 16 bytes for our experiments).

Paranoid mode: basic blocks are chopped to include only a single instruction.

Control-Flow Transfers

After binary analysis, each basic block ends with a control flow transfer instruction (e.g., jcc, jmp, call, or return and their variants). In BAST or Paranoid modes we have to insert additional transfer instructions to connect the newly chopped basic blocks. These instructions are replaced with a set of instructions that execute a lookup of the target block, transferring execution to another process if necessary. By convention, our binary analysis rewrites the basic block so that the target BID is in the `ebx` register (spilling the register if necessary). This is not optimal from a binary translation perspective but the context switching overhead to another process will dominate overhead and lookup efficiency is not a key concern.

Indirect control-flow transfer instructions like indirect jumps, indirect calls, or return instructions are harder to handle as the target BID is usually not statically known. For switch statements, IDA Pro can often recover the actual targets and replace them with the corresponding BIDs. For all remaining indirect control-flow transfer instructions we have to execute an online lookup that translates a target address to a BID. This lookup can use a

```
; if retn is used
    xchg [esp], ebx      ; backup ebx

; if retn NN is used
    mov  [esp+ARG], ebx  ; retn NN, ARG = NN*4
    mov  ebx, [esp]      ; get return address

; code for both cases
    cmp  ebx, $_RET_1
    jz   TARGET_1
    cmp  ebx, $_RET_2
    jz   TARGET_2
    ...
    mov  ebx, ffffffffh  ; ERROR
    jmp  END
TARGET_1:
    mov  ebx, $_ID_1
    jmp  END
TARGET_2:
    mov  ebx, $_ID_2
    jmp  END
    ...
END:
    nop

; if retn NN is used, remove all-1 args
    add esp, MM         ; MM = NN - 4
```

Figure 5.2.: Translation of a return instruction.

table of all target locations, or, e.g., in the case of return instructions, we can use the CFG to identify all possible call sites and encode the return targets directly in the code as follows (see Figure 5.2).

These replacements ensure that the control flow transfers are translated correctly and allow the emulator to keep executing the target binary. Any calls back to the emulator request a new target in ebx and dispatch to the next block.

Block relocations

All external references within a block must be relocated at runtime. External references can either be functions from imported modules or constant references to segments (e.g., *data*, or *rdata*). Our block metadata keeps the offset of the addresses that need runtime relocation, according with the type of relocation. In cases of indexed array accesses, or constant pointers that point to constant addresses, all we have to do is to relocate the base address.

Heap manipulation

Heap manipulation is a challenge when injecting a process into a set of benign processes as all access to the heap must be coordinated, simulating a single target address space among different host address spaces. If a block allocates memory using any of the standard heap functions, this memory will be valid only under the address space that blocks is executed. To overcome this problem we provide our own heap manipulation API, that will allocate shared memory regions at the same base address for all processes. This can be done by calling `MapViewOfFileEx()` with a non-NULL `lpBaseAddress`.

During the translation we check for heap management functions like `malloc()`, `calloc()`, `LocalAlloc()`, or `HeapAlloc()` and replace the call with an emulator-local alternative that is aware of the translation. Similar work is done for other heap management functions, like `LocalFree()` or `MapViewOfFile()`.

Socket descriptors and HANDLEs

The biggest challenge for the malWASH implementation is to transparently support HANDLEs, HKEYs (essentially a HANDLE), sockets descriptors and FILE* pointers (called "descriptors" for simplicity). Descriptors are unique per-process. If process A creates a socket, process B cannot use that socket, even if it knows the socket descriptor. However there are two functions provided by the Windows API, `DuplicateHandle()`

and `WSADuplicateSocket()` that duplicate a HANDLE and a socket respectively. Unfortunately, there is no native support for duplicating FILE* pointers. We discuss support for FILE pointers in Section 5.4.3.

Descriptor support has both an offline and an online component. Our IDA Pro plugin searches for calls to descriptor functions (complete function declaration is proviced) and marks them and their parameters for further analysis.

If a block creates, duplicates, or deletes a descriptor, this information is propagated to all other emulators using the corresponding calls. The emulator includes runtime functionality to coordinate this information.

5.4.2 Phase 2.a: Loading emulators

The loader is the first part of malWASH that executes. It initializes the required shared memory regions (administrator privileges are required to set up shared memory, obtaining these privileges is orthogonal to malWASH) and finds up to *N* processes to inject the emulator. The standard code injection involves four functions: `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. Calling these functions in that order is suspicious.

Although we cannot avoid to call these functions in that order, we make detection harder in two ways. The first is to recursively use the chopping idea of malWASH: the loader spawns 3 new processes. Each of these processes calls exactly one of the four functions and informs the next one to continue. HANDLEs can be duplicated using DuplicateHandle() and shared with any Inter Process Communication (IPC) mechanism. This does not solve the problem but it adds one more layer of indirection. The second way we make detection more difficult is to use equivalent undocumented functions from the NT API: `ZwOpenProcess`, `ZwAllocateVirtualMemory`, `ZwWriteVirtualMemory`, and `NtCreateThreadEx`. Both `CreateThread` (a benign function) and `CreateRemoteThread` (a notorious function), internally call `NtCreateThreadEx`. Thus a detection tool has to check the function arguments to decide if a call is malicious or not, resulting in performance overhead.

If these mitigations are not enough, the loader can spawn new processes, instead of infecting existing ones, or infect non-running processes using one of the existing methods viruses use for injection [145]. These approaches are not a panacea against detection and we assume that the loader is, for now, trusted.

5.4.3 Phase 2.b: Executing the binary

After the loader finishes, it exits, and the emulator starts executing the individual pieces of program code, emulating a regular process environment. The emulator runs under a foreign process, like a parasite, and has no knowledge of the environment during start up. This makes the development of the emulator an extremely challenging process. Written in pure assembly, the emulator consists of 5,500 lines of assembly code (less than 14 kB of compiled code) and can execute all the blocks in the correct order.

Core environment

When the emulator starts executing it must first establish its execution environment. By reading the Process Environment Block (PEB), the emulator finds the entry point of kernel32.dll and the address of `LoadLibrary()` and `GetProcAddress()`, allowing us to find all other addresses in the system. The emulator then queries for a (randomly) named shared memory region that contains the emulator state and the shared heap.

To get to an executable state, constant addresses to segments must be relocated to shared regions and functions must be resolved to actual addresses, except some special functions (e.g., those in Section 5.4.1) that are redirected to internal functions of the emulator.

The emulator keeps “virtual registers” that the original binary will use. Context switching is done before and after block execution. In each iteration the emulator waits on a semaphore to get a mutual lock to execute the next block. When it takes the lock it copies the next block into a local buffer. Eventually, the emulator will start executing the block using the virtual registers. When the block finishes, the `ebx` register will contain the next BID and control returns to the emulator to dispatch the next block.

There is also a special shared region, called *Shared Control Region*. This region coordinates all emulators and contains (among other fields) the virtual registers. Stack is handled like other segments. During startup, the emulator sets the virtual `esp` and `ebp`, with the value of the shared stack, so the malware will not see any difference and will use the shared stack instead. The loader prepares any command line arguments of the original program on the stack.

Advanced Components

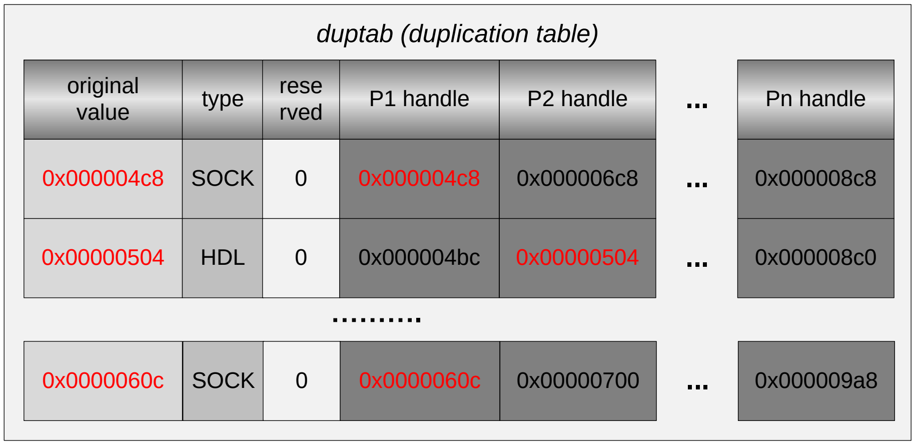
So far, the emulator can execute a program under multiple address spaces but there are many small details that may cause the execution to fail. Here we discuss and address these problems.

All emulators need to communicate. We therefore reserve some space in the shared control region and use it as a mailbox. Emulators communicate by sending messages, each message consists of a header followed by the data (a message looks like an UDP packet). Emulators check their mailboxes (e.g., they simply read the value of the mail counter) and process any messages, before execution of each block.

Section 5.4.1 discussed the challenge of duplicating descriptors between emulators. The offline part replaces the use of the descriptors with calls into the emulator. Here, we discuss the implementation of these functions. We allocate a table (called Duplication Table or "duptab") with function pointers for each of the internal descriptor functions and dispatch the functions accordingly. An instance of duptab is shown in Figure 5.3.

Duptab contains one row for each descriptor that the original binary uses. Each row contains the original value, the type (socket, HANDLE, or HKEY) of the descriptor, and the value of the duplicated descriptor for each host process. The emulator functions then use this table to translate a descriptor to the local descriptor.

Unfortunately, there is no mechanism to duplicate FILE* pointers. We solve this problem by using an alternative approach: We provide our API replacements for functions that use FILE* pointers. These replacements are simple wrappers of equivalent functions that use



duptab (duplication table)

original value	type	rese rved	P1 handle	P2 handle	...	Pn handle
0x000004c8	SOCK	0	0x000004c8	0x000006c8	...	0x000008c8
0x00000504	HDL	0	0x000004bc	0x00000504	...	0x000008c0
....						
0x0000060c	SOCK	0	0x0000060c	0x00000700	...	0x000009a8

Figure 5.3.: An instance of duptab

HANDLEs (which we can duplicate). E.g., `fopen()`, is a wrapper for `CreateFileW()`, `fprintf()` is a wrapper for `sprintf()` and `WriteFile()` and so on.

Beyond FILE* functions, several other functions need replacement. For instance, if the original binary calls `ExitProcess()`, we terminate all emulators (instead of terminating the current process). The emulator keeps a list of such special functions and replaces them with the internal implementations during startup. Other types of functions that need replacements are: functions that perform per-process specific actions (e.g., `SetCurrentDirectory()`) or functions that keep internal state (e.g., , `strtok()`). In both cases, the emulator has to replicate the information across all emulator instances.

There are some sequences of functions, that must be executed in the same address space, e.g., `{bind, listen, accept}` and `{GetStartupInfo, CreateProcess}`. If `listen()` is executing in a different address space than `bind()`, even though the socket is successfully duplicated, an WSAEINVAL error will be returned (this is a Windows bug). Our emulator uses a *call cache* to address this issue. Each function in a chain is marked as *push* while the last one is marked as *sweep*. Replacements are provided for these functions to include the push-sweep functionality. An emulator does not execute a *push* function; instead it pushes the function (with its arguments) on the *call cache* and returns a fake successful value. When an emulator finds a *sweep* function it executes all functions from the *call cache* along with the last one, flushing the *call cache*. Although not perfect, this approach works well in practice.

The distributed design of malWASH allows us to handle multi-threaded programs by creating a shared stack and virtual registers for each thread. Each thread contains its own semaphore and its own variable that indicates the next block. Each emulator uses a round-robin algorithm to execute blocks from all "RUNNING" threads. Simple replacements are also provided for thread management functions: `CreateThread()` and `ResumeThread()` mark and emulated thread as "RUNNING", `ExitThread()` marks it as "UNUSED" and `SuspendThread()` marks it as "SUSPENDED".

The job of the emulator is twofold; it executes the emulated binary and keeps itself stealthy. Emulators can "ping" other emulators to see if all of them are alive. When an emulator detects that some are missing, it could invoke the loader to inject the missing emulator into a new process.

5.4.4 Recovering terminated instances

The core functionality of malWASH is to ensure that the original binary executes as if being run in a regular environment. In addition, malWASH also ensures resilience and recovery against "attacks".

Resilience, is enforced as a side effect of malWASH's distributed nature. We may run into the problem that an analyst kills all but one emulator instances to simplify the analysis process. Therefore, malWASH also needs a *recovery* mechanism. We already have a communication mechanism between emulators (Section 5.4.3) and as we mentioned in Section 5.3, the total number of running emulators is constant and known to all emulators. Thus, checking whether an emulator was killed is straight forward: each emulator periodically sends *heartbeat* messages to all emulators. If an emulator stops receiving heartbeats, it can invoke the loader process again, to respawn the missing emulators.

Table 5.2.: Block statistics of malware samples.

Sample name	Type	## Instructions	Blocks Generated		
			BBS	**BAST**	**Paranoid**
Trojan.Win32.Keylogger.Gen	keylogger	2957	347	541	1484
Trojan.Win32.Invader.aa	backdoor	6359	118	233	782
Gen:Heur.Bodegun.8	backdoor	1326	112	195	496
Virus.Win32.FileInfector	virus	1739	98	183	772
TrojanSpy:Win32/Keylogger.BZ	keylogger	1380	89	178	546
Trojan-Spy.Win32.DiabloII.a	trojan-spy	162	62	86	162
W32/S-ac5b79f0!Eldorado	trojan	1837	67	141	431
W32/SelfStarterInternetTrojan!M	trojan-backdoor	3391	107	209	576

5.5 Evaluation

We evaluate malWASH by targeting a set of malware samples that we inject into the most popular browsers (Google Chrome v50.0.2661.94, Mozilla Firefox 6.0.1 32 bit, Opera 12.16 and Safari 5.1.7) as victim processes under the Windows 8.1 Pro x64 operating system. Chrome's security feature of separating each tab as its own process comes in handy and allows malWASH to inject a different set of chunks into each per-tab process and shared memory regions across Chrome instances will not raise alarms.

Table 5.2 shows details of the malware samples we evaluate. The total number of instructions is not equal to the number of blocks in paranoid mode as malWASH omits code before and after `main()` as the malWASH loader component sets up the process environment and not the initialization code in the executable.

We inject malWASH into 1, 2, 4, and 8 Chrome processes, executing the samples in the different modes. In all cases both the host processes and the emulated process run without error. The host process continues without measurable performance degradation.

5.5.1 malWASH resilience

Due to the distributed nature and the shared state of malWASH, killing an emulated process is hard. In Figure 5.4 we inject a sample into 8 idle processes (so any CPU usage will come from malWASH) and start measuring their CPU usage using Microsoft Performance

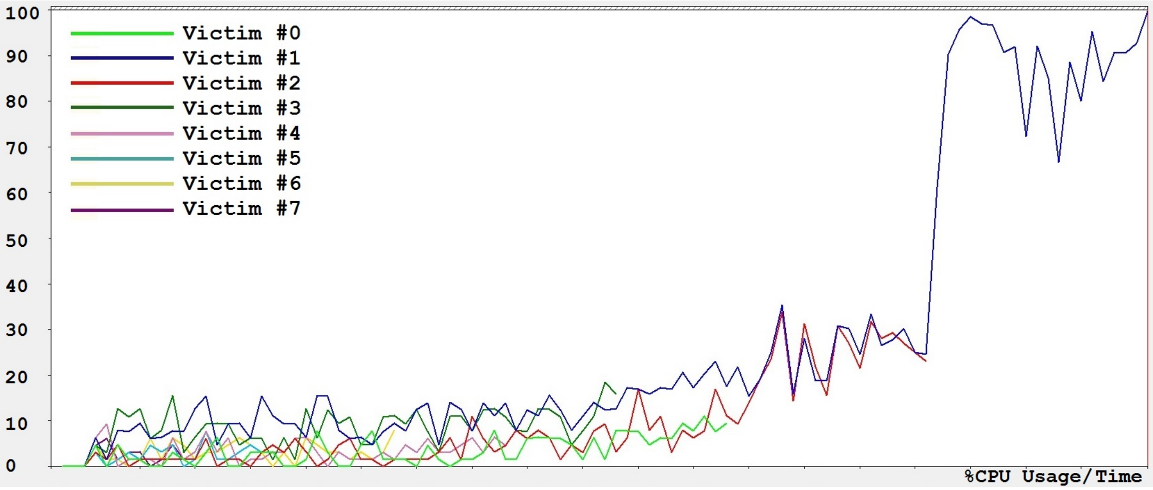


Figure 5.4.: CPU usage among infected (idle) processes

Monitor. Initially, all host processes execute roughly the same number of blocks, so the CPU per host process stays low. As we kill off individual host processes, the remaining emulators end up executing more blocks, increasing their CPU usage. If additional stealth is required, the emulators can throttle execution of the target process and add sleep intervals between block executions.

5.5.2 Case Study: Remote Keylogger

For malWASH we assume that the target process is not CPU intensive. For CPU intensive workloads, the emulator may be an issue as there is overhead between executed blocks. Our emulator works well for programs that require stealthiness with little computation. Examples of such programs are keyloggers or host-based backdoors. In this section we focus on a remote keylogger to demonstrate the effectiveness of malWASH.

The remote keylogger works a follows: it opens a TCP connection to a remote host and sends captured keystrokes to the host. For the evaluation, the keystrokes were sent to a different process on the same machine. The target program is repeatedly checking whether the foreground window contains keywords from a whitelist (e.g., Facebook, GMail, Hotmail, or Twitter). And if so, it starts keylogging by checking the state of each key.

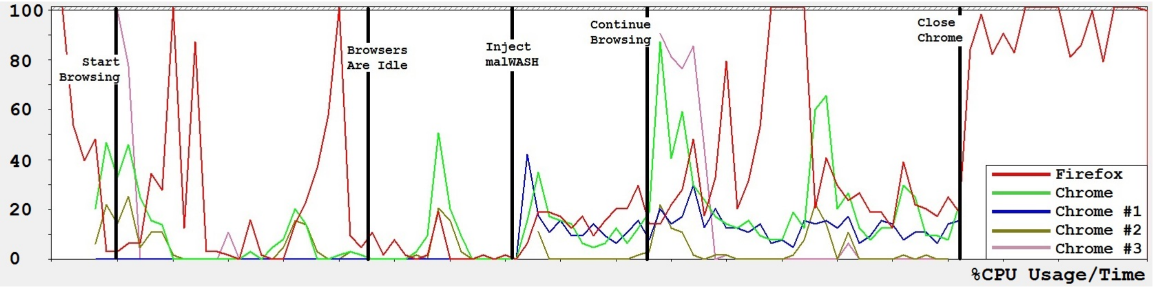


Figure 5.5.: CPU usage of Firefox and Chrome under malWASH infection

We measured performance impact by using the Octane 2.0 JavaScript benchmark on the host browsers' processes. In this benchmark we inject malWASH into the browser process that runs the benchmark for each experiment. Table 5.3 shows the average and standard deviation of the benchmark scores for five runs, the low standard deviation shows that the results are stable. The difference of the performance results across injected and non-injected version is in the noise and will make intrusion detection based on performance results hard.

Figure 5.5 shows a second scenario where we inject the keylogger under malWASH in one Firefox process and four Chrome processes (Chrome has four running processes even with a single open tab), measuring their CPU usage using the Microsoft Performance Monitor. During normal browsing we observe some spikes due to regular browsing activity. Then we stop browsing (browsers are idle) and inject malWASH. At this point there is a small peak due to malWASH startup. As browsing continues, the keylogger now runs inside the host processes and captures keystrokes. After some time we close Chrome and the emulator inside Firefox now has to execute all blocks, showing a slight increase in CPU usage for the Firefox process.

This benchmark shows that we can distribute the load of the emulator across several processes. With an increasing amount of host processes, the overhead for each individual host process through the injected process is reduced.

Table 5.3.: Statistics from running the Octane 2.0 JavaScript benchmark five times in each of the most popular browsers, “w/o” shows execution without injection, “Std” shows a keylogger that scans for keywords for half of the time and captures and sends keystrokes for the rest of the time, while “Full” shows the keylogger capturing and sending keystrokes 100% of the time.

Average scores from Octance 2.0 Javascript Benchmark												
	Google Chrome			*Mozilla Firefox*			*Opera*			*Safari*		
Mode	w/o	Std	Full	w/o	Std	Full	w/o	Std	Full	w/o	Std	Full
Average	19,541	15,762	11,226	16,259	12,146	10,356	6,048	4,832	3,988	3,163	2,328	2,041
St. Dev	316	754	1,431	947	2.727	650	201	250	136	99	153	38

5.5.3 Discussion

Detecting programs running under malWASH through static or dynamic analysis is difficult. Static analysis is complicated because the original binary is chopped into many small pieces, likely below the signature threshold. The (tiny) emulator itself can also be protected using existing (automated) diversity techniques. Dynamic analysis is challenging as the behavior of the target program is hidden under the infected processes, making it hard to observe a sequence of calls of the target program. Therefore, defenders will likely move towards detecting malWASH instead of the target program. This by itself has the advantage of hiding the true functionality of the emulated program.

Protecting the emulators

Although existing detection methods will have a hard time detecting the original binary, they can be used for detecting the emulators. We argue that behavioral analysis of emulator is challenging because: (i) the emulator is very small (14kB), (ii) the emulator uses only a tiny set of system calls (for shared memory management) which will appear benign, and, most importantly, (iii) these system calls are well mixed with a subset of system calls from the emulated binary. In addition, the emulator can leverage any existing obfuscation techniques to make analysis harder.

An issue that the emulator faces is that it uses dedicated threads with similar behavior. Thus, instead of a per-process analysis, a defender could look at the actual threads and try to identify emulator threads. However, this situation is somewhat similar to the status quo: malware uses a dedicated process within the system. One option would be to chop the emulator itself into small components, injecting them into different threads of the same process. This would lead to yet another (smaller) sub-emulator. Sub-emulators are much simpler because they run under the same address space and thus they lack the aforementioned problems that malWASH tries to solve. No shared memory is required, just a form of synchronization (e.g., spin locks or covert channels), hardening the options for behavioral analysis and spreading the emulator across several threads.

Fixing any abnormal system behavior

The performance overhead for malWASH is small for non-CPU intensive workloads (see Figure 5.4). A possible detection mechanism could spread "honeypot" processes that are idling on the system. As soon as the emulator is injected into these processes they will start to execute *some* computation and the malWASH injection can be detected. malWASH can try to mitigate by scanning for active processes by making the loader more complex (and therefore more detectable).

Careful selection of host processes, hides potential behavioral discrepancies of a process, e.g., no alarms are raised for an emulator that opens a remote connection if it is running in a browser. Process selection is an open problem and we leave it as a future work. In short, malWASH could observe the behavior of a process, and if suitable, do the injection.

Another opportunity to detect malWASH is the shared memory regions. A detection mechanism may correlate host processes through their shared memory regions. On one hand correlation is challenging, due to the large amount of shared memory regions that are active across all processes on windows systems. In addition, malWASH does not require a star-like mapping where the same shared memory region is mapped among all processes (even for heap allocated shared regions) but can also use duplicated regions as shown in Figure 5.6.

With duplicated regions, we maintain multiple copies of the same shared mapping, and we force at most two processes to share the same region. Each region could then use a disjoint encryption key to avoid correlation between shared regions. In order to keep these shared regions consistent, some "external" processes are needed. Each external process is responsible for keeping the subset of shared regions consistent. External processes communicate with each other to keep their subsets consistent. This communication is done using covert channels or by reading/writing regions to temporary files to avoid "circles" of processes connected by shared memory.

In case that usage of shared memory is a problem by itself, it can safely replaced by different (and admittedly slower) mechanisms like files, pipes, or covert channels.

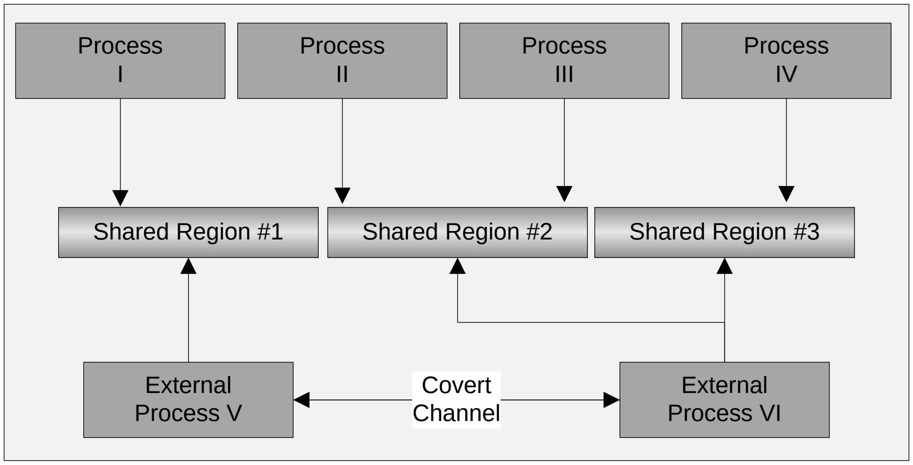


Figure 5.6.: Thwarting detection based on shared memory correlation. Here processes I through IV used to share the same mapping. We create 3 replicas for the shared mapping with two processes attached each.

Also, the distributed nature of malWASH does not require all the blocks and program's state to be present in memory during execution: the emulator could request the next block and the current program's state from a remote host which is controlled by the bot of the attacker.

As discussed in Section 5.4.2, the loading is the most exposed part of malWASH. If our proposed obfuscation approach is not stealthy enough, additional emulator processes can be spawned on demand, further obfuscating the loader.

We do not claim that this section covers all methods to detect malWASH and other ways may exist. The current prototype of malWASH is not complete but focuses on showcasing the technique. Overall, malWASH is a new technique to hide a target program in a set of benign processes.

5.6 Conclusion

Hiding processes in an execution environment is a challenging problem. While static detection is straight-forward to evade using metamorphism [26] and diversity, dynamic detection can single out processes at runtime due to their behavior.

We present malWASH, a tool that hides the behavior of an arbitrary program by distributing the program's execution across many processes. We break the program into small chunks and inject these chunks into other processes. Our emulator captures and synchronizes state among the processes and coordinates the execution of the program, hopping from process to process and weaving individual instructions and system calls into the stream of instructions and system calls of the host program. We also propose the use of sub-emulators to further protect malWASH itself.

Our evaluation shows that our prototype of malWASH successfully distributes different malware programs into sets of benign processes. Detecting coordinated small chunks of malicious code in benign processes is a challenging problem for the research community.

6 RELATED & FUTURE WORK

As discussed in Chapter 1, precisely inferring the Residual Attack Surface, is not possible as it is based on an undecidable problem (refer to Appendix 7.1 for the proof). Approximating it remains an open problem with lots of interesting directions to explore. This dissertation explores only a small portion of it (e.g., malWASH is just one way to achieve persistence on a compromised system while evading detection). Finding new attacks is beneficial for defenders as they can reinforce their defense mechanisms.

6.0.1 Library Fuzzing

Fuzzing remains the most widely deployed technique for discovering new vulnerabilities. One major factor of its widespread use is simplicity: the target application is fed with some random input while fuzzer looks for abnormal behavior (crashes, or hangs). However, scaling fuzzing to libraries is challenging, as libraries are not standalone applications with a well defined entry point. Existing solutions include *i)* libFuzzer [53] and *ii)* fuzzing standalone programs (called *consumers*) that utilize API functions from this library. Although libFuzzer provides a convenient way to fuzz individual API functions, it involves a huge amount of manual effort, as the analyst needs to figure out how to fuzz the individual functions. Fuzzing library a consumer has some major limitations too. First, consumers may explore only a small portion of the library. Second, it is hard to determine whether the discovered bugs are from the library and not the consumer itself.

FuzzGen is the first attempt to solve this problem. However it can be improved in several directions. Analysis can be imprecise or even fail in some cases, so improving the analysis is our first goal. Furthermore, the generated fuzzers are heavily dependent on the library consumers. Having too many consumers, results in cumbersome and slower fuzzers. Also, when the consumers explore only a small portion of the library, FuzzGen produces weak fuzzers. Classifying library consumers and selecting an appropriate subset is also the next topic of our future work.

6.0.2 Data-Only and Control Flow Bending attacks

With the wide deployment of CFI, ROP is no longer possible. Nevertheless, it is still possible to perform code reuse attacks [15, 97, 98, 99], as well as data-only attacks [16, 17, 100, 133]. However, these attacks are extremely hard in practice as they have requirements (e.g., arbitrary memory write primitives) that are hard to find. Even worse, it has been shown [17] that the problem of automating data-only attacks is reduced to an NP-hard problem. Hence, research in this areas seems to be saturated, as there are not many new things to explore. A potential extension is to include the applied CFI policy in BOPC framework, to include the likelihood of finding a solution when a coarse-grained CFI policy is applied.

On the other hand, preventing data-only attacks remains an open problem as existing DFI protection schemes come with a high overhead. During the evaluation of BOPC, we noticed that it is possible to insert additional code in the binary, that clobbers a set of given basic blocks. Thus finding dispatcher gadgets to stitch functional blocks together is no longer possible. Finding a way to thwart BOP gadget stitching with a low overhead is an interesting challenge that we will look into.

6.0.3 Distributed malware detection

Detecting malware through dynamic and behavioral analysis is an effective measure against obfuscated and metamorphic malware where static analysis fails. Although it is easy to change the shape of a malware, it is hard to change its identity and its intentions. With the concept of distributed malware [37], attackers can “mix” the behavior of the malware with other processes on the system thus bypassing existing detection mechanisms. Detecting this form of distributed malware is an interesting challenge, as existing detection mechanisms are not designed to scale to multiple processes. Finding a new detection scheme that is capable of analyzing two or more processes at the same time with low overhead is also an interesting problem to look into.

7 CONCLUSION

This dissertation presented the body of the work on infering the *Residual Attack Surface* under state-of-the-art mitigations. The dissertation started with a definition of the Residual Attack Surface and continued with the challenges in measuring it. The key insight was to divide an attack into distinct phases (*Vulnerability Discovery*, *Vulnerability Exploitation*, *Persistence on the compromised system*) and to infer the Residual Attack Surface in each phase.

FuzzGen is a tool for automatically synthesizing target-specific fuzzers that able to achieve a high code coverage and hence expose bugs that reside deep in the code. FuzzGen is part of the *Vulnerability Discovery* phase and assist software developers to quickly find and patch bugs before an attacker exploits them.

BOPC [17] is a framework that implements the concept of *Block Oriented Programming* which automates Data-Only attacks under heavily constrained environments such as binaries hardened with CFI and shadow stacks. BOPC is part of the *Vulnerability Exploitation* phase and can help software developers to highlight payloads that an attacker is still capable of executing.

An extension of BOPC is *X-Cap*, which is an ongoing work. It essentially assesses exploitation capabilities by indicating what types of payloads are feasible to run in vulnerable applications. X-Cap highlights the limits of BOPC and provides *upper bounds* on attacker's capabilities.

malWASH [37] is another framework for the last phase of the attack that thwarts dynamic and behavioral analysis to achieve persistence on the compromised system. malWASH automatically "chops" a binary into hundreds of piece and performs a *distributed execution* on them. malWASH can help malware analysts to evaluate their detection tool and include potential detection schemes for distributed malware in their defense mechanisms.

In conclusion, we hope that the Residual Attack Surface will lead to new defenses. These defenses should be adapted to the new attack technologies and possibilities that attackers invent to bypass existing mitigations.

APPENDIX

7.1 Determining exploitability is undecidable

We present a proof that the problem of determining the exploitability of a security bug (i.e. vulnerability) is undecidable. We prove this statement by *contradiction*, by reducing halting [146] problem to it.

Let us assume that it is possible to determine whether a vulnerability is exploitable. In this case there should exist a Turing machine $EXPL_M$, that decides (i.e., always terminates) whether another Turing machine M (i.e., a program) with an known vulnerability, is exploitable when running on some given input w. $EXPL_M$ is formally defined as follows:

$$EXPL_M(M, w) = \begin{cases} accept, & \text{if running } M \text{ on } w \text{ exploits a vulnerability} \\ reject, & \text{otherwise} \end{cases} \tag{7.1}$$

Given $EXPL_M$, we will build a Turing machine $HALT_M$ that determines whether another Turing machine M terminates (halts) when running on some input w:

$$HALT(M, w) = \begin{cases} accept, & \text{if running } M \text{ on } w \text{ terminates} \\ reject, & \text{otherwise} \end{cases} \tag{7.2}$$

Let also M' be a Turing machine that operates on three distinct inputs: The *description* $\langle M \rangle$ of another Turing machine M, some input w, and some exploit payload x. The description of M' is the following:

$M' =$ ‘ for input $(\langle M \rangle, w, x)$‘

1. Run $EXPL_M$ on (M, w)
2. if it accepts, then $reject$
3. Simulate M on w

```
    4. If $M$ $accepts$, or $rejects$, then
    5. Trigger the vulnerability and execute payload $x$
```

Having all these components, we can build a Turing machine $HALT_M$ that decides whether a Turing machine terminates:

```
$HALT_M =$ ‘ for input $(\langle M \rangle, w)$‘
    1. Run $EXPL_M$ on $M'$ with input $(\langle M \rangle, w, x)$
    2. If it accepts, then $accept$
    3. Otherwise $reject$
```

The intuition behind M' is that, if M does not terminate with input w, then it will never reach step 5 and hence it will never exploit a vulnerability. Thus, $EXPL_M(\langle M \rangle, w, x)$ will $reject$. On the other hand, if M terminates when running with input w after a finite number of steps, then M' will reach step 5 which means that the M' will trigger the vulnerability and execute a payload. This means that M' has an exploitable vulnerability and therefore $EXPL_M(\langle M \rangle, w, x)$ $accepts$ input. However, there's a special case. What if running M on w exploits an vulnerability itself? In that case, $EXPL_M$ will accept M', even if the exploit payload does *not* terminate. We do not consider this case, as we assume that M does not have any vulnerabilities.

The above statement indicates that it is possible to build $HALT_M$ from $EXPL_M$. This implies that we have a solution for the *halting problem.* This is of course is not possible. Therefore, the initial assumption (i.e., *it is possible to determine whether vulnerability is exploitable*) contradicts with our result. Thus, $EXPL_M$ cannot exist and hence the problem of determining exploitability is undecidable.

7.2 State Inconsistency for A^2DG coalescing

Although coalescing increases the generality of the fuzzers, it suffers from the *state inconsistency* problem. Consider for instance a fuzzer of a socket library and two library consumers (a) and (b) as shown below:

```
/* consumer #1 */
sd = socket(...);
connect(...);

// send only sock
shutdown(sd,
         SHUT_RD);
write(sd, ...);

close(sd);
```

(a)

```
/* consumer #2 */
sd = socket(...);
connect(...);

// send & recv
write(sd, ...);

read(sd, ...);

close(sd);
```

(b)

```
/* coalesced */
sd = socket(...);
connect(...);

shutdown(sd,
         SHUT_RD);
write(sd, ...);
read(sd, ...);

close(sd);
```

(c)

The first module connects to a server and terminates the read side of the socket (as it only sends data). The second module both sends and receives data. If we ignore the arguments for now, the functions `socket`, `connect` and `write` are shared between the two consumers and they are therefore coalesced. The result is the coalesced fuzzer shown in (c). However this results in an inconsistency where the fuzzer closes the read side of the socket and later tries to read from it. Although the fuzzer does not crash, the coalesced module violates the state and is therefore not a useful fuzzer.

A^2DG coalescing results in aggressive fuzzers that achieve deeper coverage and find more bugs. The downside is that coalescing may introduce false positives where the API is violated, resulting in false bugs. Without coalescing, the fuzzers are redundant and will not achieve coverage as deep as the coalesced fuzzers but will not introduce any false positives. In our empirical evaluation we discovered that the number of false positives is low and we therefore enable coalescing but leave it as a configurable option. In future work, we will look at how to tune coalescing aggressiveness, i.e., deciding how and when to coalesce based on a heuristic.

7.3 Extended Backus-Naur Form of SPL

```
⟨SPL⟩ ::= void payload( ) { ⟨stmts⟩ }
⟨stmts⟩ ::= (⟨stmt⟩ | ⟨label⟩)* ⟨return⟩?
⟨stmt⟩ ::= ⟨varset⟩ | ⟨regset⟩ | ⟨regmod⟩ | ⟨call⟩
        |  ⟨memwr⟩ | ⟨memrd⟩ | ⟨cond⟩ | ⟨jump⟩

⟨varset⟩ ::= int64 ⟨var⟩ = ⟨rvalue⟩;
        |  int64* ⟨var⟩ = {⟨rvalue⟩ (, ⟨rvalue⟩)*};
        |  string ⟨var⟩ = ⟨str⟩;
⟨regset⟩ ::= ⟨reg⟩ = ⟨rvalue⟩;
⟨regmod⟩ ::= ⟨reg⟩ ⟨op⟩= ⟨number⟩;
⟨memwr⟩ ::= *⟨reg⟩ = ⟨reg⟩;
⟨memrd⟩ ::= ⟨reg⟩ = *⟨reg⟩;
⟨call⟩ ::= ⟨var⟩ ( ( ϵ | ⟨reg⟩ (, ⟨reg⟩)* );
⟨label⟩ ::= ⟨var⟩:
⟨cond⟩ ::= if (⟨reg⟩ ⟨cmpop⟩ ⟨number⟩) goto ⟨var⟩;
⟨jump⟩ ::= goto ⟨var⟩;
⟨return⟩ ::= returnto ⟨number⟩;

⟨reg⟩   := ‘__r’⟨regid⟩
⟨regid⟩ := [0-7]
⟨var⟩   := [a-zA-Z_][a-zA-Z_0-9]*
⟨number⟩ := (‘+’ | ‘−’) [0-9]+ | ‘0x’[0-9a-fA-F]+
⟨rvalue⟩ := ⟨number⟩ | ‘&’ ⟨var⟩
⟨str⟩   := [.]*
⟨op⟩    := ‘+’ | ‘−’ | ‘⋆’ | ‘/’ | ‘&’ | ‘|’ | ‘~’ | ‘«’ | ‘«’
⟨cmpop⟩ := ‘==’ | ‘!=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’
```

7.4 Stitching BOP Gadgets is NP-Hard

We present the NP-hardness proof for the BOP Gadget stitching problem. This problem reduces to the problem of finding the *minimum induced subgraph* H_k in a delta graph. Furthermore, we show that this problem cannot even be approximated.

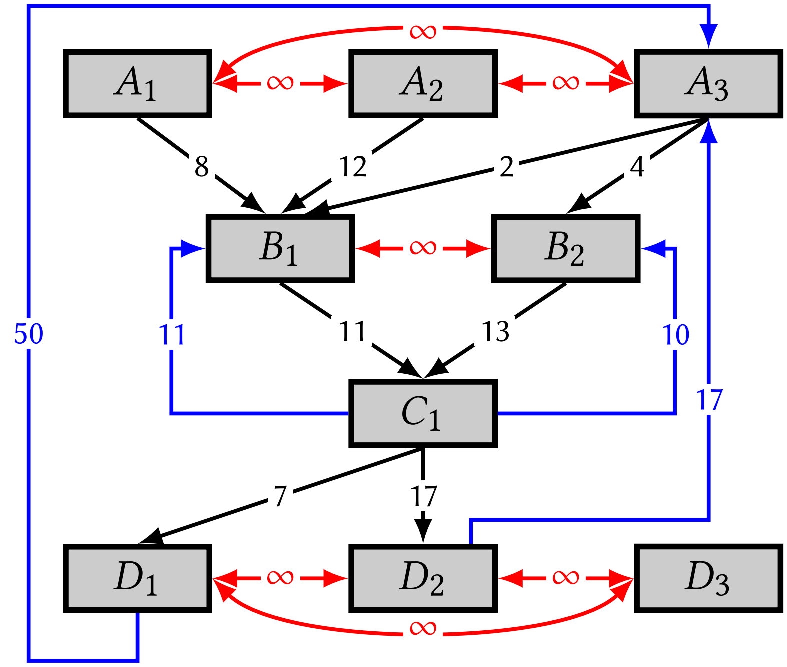


Figure 7.1.: An delta graph instance. The nodes along the black edges form a *flat* delta graph. In this case, the *minimum induced subgraph*, H_k is A_3, B_1, C_1, D_1, with a total weight of 20, which is also the *shortest path* from A_3 to D_1. When delta graph is *not* flat (assume that we add the blue edges), the shortest path nodes constitute an induced subgraph with a total weight of 70. However H_k has total weight 34 and contains A_3, B_2, C_1, D_2. Finally, the problem of finding the minimum induced subgraph becomes equivalent to finding a k-clique if we add the red edges with ∞ cost between all nodes in the same set.

Let δG be a *multipartite* directed weighted delta graph with k sets. Our goal is to select *exactly* one node (i.e., functional block) from each set and form the *induced subgraph* H_k, such that the total weight of all of edges is *minimized*:

$$\min_{H_k \subset \delta G} \sum_{e \in H_k} distance(e) \tag{7.3}$$

A δG is *flat*, when all edges from i^{th} set are towards $(i+1)^{th}$ set. The nodes and the black edges in Figure 7.1 are such an example. In this case, the minimum induced subgraph, is the minimum among all *shortest paths* that start from some node in the first set and end in any node in the last set. However, if the δG is *not* flat (i.e., the SPL payload contains jump statements, so edges from i^{th} set can go anywhere), the shortest path approach does not work any more. Going back in Figure 7.1, if we make some loops (add the blue edges), the previous approach does not give the correct solution.

It turns out that the problem is NP-hard if the δG is not flat . To prove this, we will use a reduction from ***K-Clique***: First we apply some equivalent transformations to the problem. Instead of having K independent sets, we add an edge with ∞ weight between every pair on the same set, as shown in Figure 7.1 (red edges). Then, the minimum weight K-induced subgraph H_k, cannot have two nodes from the same set, as this would imply that H_k contains an edge with ∞ weight.

Let R be an undirected un-weighted graph that we want to check whether it has a k-clique. That is, we want to check whether $clique(R,k)$ is True or not. Thus, we create a new directed graph R' as follows:

- R' contains all the nodes from R
- $\forall$ edge $(u,v) \in R$, we add the edges (u,v) and (v,u) in R' with $weight = 0$
- $\forall$ edge $(u,v) \notin R$, we add the edges (u,v) and (v,u) in R' with $weight = \infty$

Then we try to find the *minimum weight k-induced subgraph* H_k in R'. It is true that:

$$\sum_{e \in H_k} weight(e) < \infty \Leftrightarrow clique(R,k) = True$$

$:\Rightarrow$ If the total edge weight of H_k is not ∞, this implies that for every pair of nodes in H_k, there is an edge with weight 1 in R' and thus an edge in R. This by definition means that the nodes of H_k form a k-clique in R. Otherwise (the total edge weight of H_k is ∞) it means that it does not exist a set of k nodes in R' that has all edge weights $< \infty$.

:⇐ If R has a k-clique, then there will be a set of k nodes that are fully connected. This set of nodes will have no edge with ∞ weight in R'. Thus, these nodes will form an induced subgraph of R' and the total weight will be smaller than ∞.

This completes the proof that finding the minimum induced subgraph in δG is NP-hard. However, no (multiplicative) approximation algorithm does exists, as it would also solve the K-Clique problem (it must return 0 if there is a K-Clique).

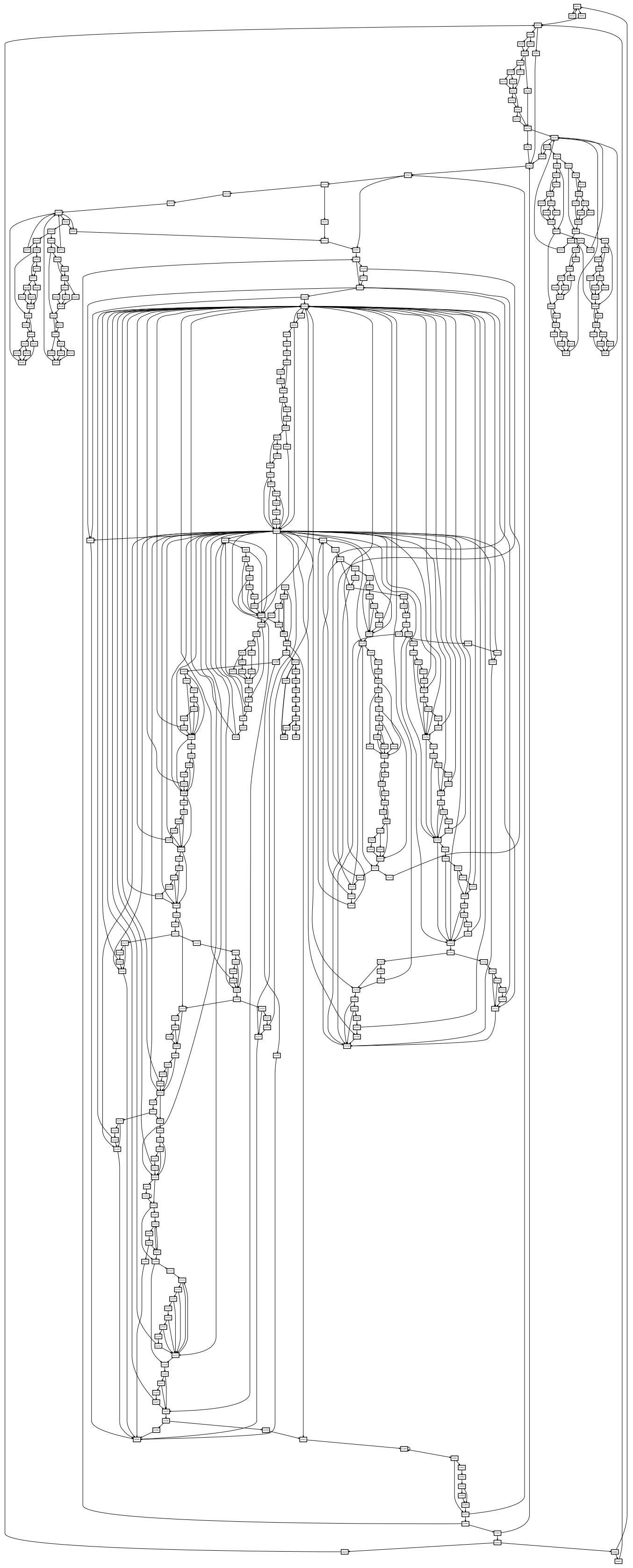
7.5 SPL is Turing-complete

We present a constructive proof of Turing-completeness through building an interpreter for Brainfuck [147], a Turing-complete language in the following listing. This interpreter is written using SPL with a Brainfuck program provided as input in the SPL payload.

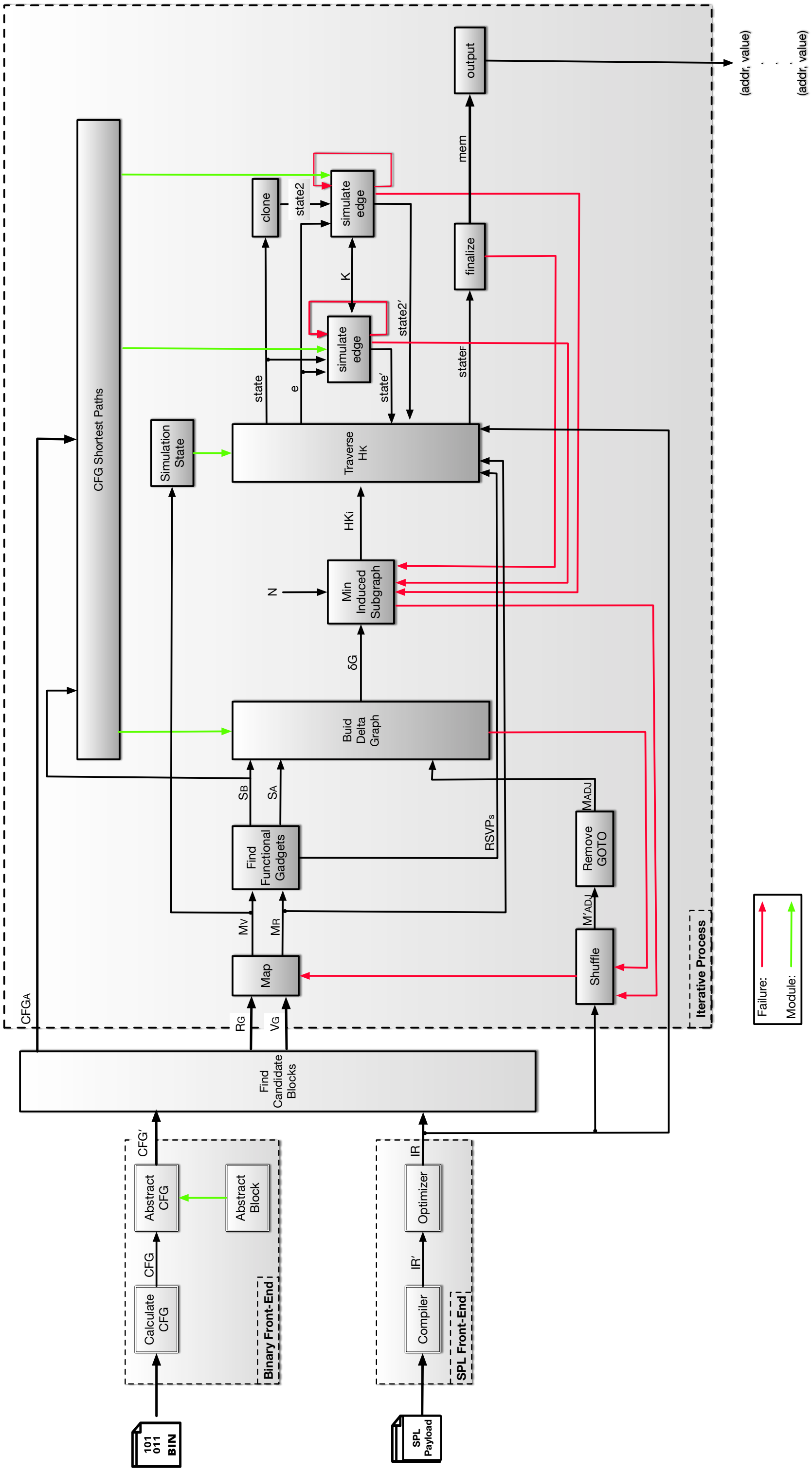
```
int64 *tape  = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
string input = ".+[.+]";
__r0 = &tape;      // Data pointer
__r2 = &input;     // Instruction pointer
__r6 = 0;          // STDIN
__r7 = 1;          // STDOUT
__r8 = 1;          // Count arg for write/read
NEXT:    __r1 = *__r2;
         if (__r1 != 0x3e) goto LESS;    // '>'
         __r0 += 1;
LESS:    if (__r1 != 0x3c) goto PLUS;    // '<'
         __r0 -= 1;
PLUS:    if (__r1 != 0x2b) goto MINUS;   // '+'
         *__r0 += 1;
MINUS:   if (__r1 != 0x2d) goto DOT;     // '-'
         *__r0 -= 1;
DOT:     if (__r1 != 0x2e) goto COMMA;   // '.'
         write(__r7, __r0, __r8);
COMMA:   if (__r1 != 0x2c) goto OPEN;    // ','
         read(__r6, *__r0, __r8);
OPEN:    if (__r1 != 0x5b) goto CLOSE;   // '['
         if (__r0 != 0) goto CLOSE;
         __r3 = 1;        // Loop depth counter
FIND_C:  if (__r3 <= 0) goto CLOSE;
         __r2 += 1;
         __r1 = *__r2;
         if (__r1 != 0x5b) goto CHECK_C; // '['
         __r3 += 1;
CHECK_C: if (__r1 != 0x5d) goto FIND_C;  // ']'
         __r3 -= 1;
         goto FIND_C;
CLOSE:   if (__r1 != 0x5d) goto END;     // ']'
         if (__r0 != 0) goto END;
         __r3 = 1;        // Loop depth counter
FIND_O:  if (__r3 <= 0) goto END;
         __r2 -= 1;
         __r1 = *__r2;
         if (__r1 != 0x5b) goto CHECK_O; // '['
        __r3 -= 1;
CHECK_O: if (__r1 != 0x5d) goto FIND_O;  // ']'
        __r3 += 1;
        goto FIND_O;
END:    __r2 += 1;
goto NEXT;
```

7.6 CFG of nginx after pruning

The following graph, is a portion of nginx's CFG that includes function calls starting from the function `ngx_cache_manager_process_cycle`. The graph only displays functions which are up to 3 function calls deep to simplify visualization. Note the reduction in search space–which is a result of BOPC's pruning–as this portion of the CFG reduces to the small delta graph in Figure 3.7.



7.7 Detailed overview of the BOPC implementation



Bibliography

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 48–62.

[2] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, 1976.

[3] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[4] H. Peng, Y. Shoshitaishvili, and M. Payer, “T-fuzz: fuzzing by program transformation,” 2018.

[5] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[6] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, “Automatic exploit generation,” *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[7] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, “Revery: From proof-of-concept to exploitable,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1914–1927.

[8] A. van de Ven and I. Molnar, “Exec shield,” https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[9] S. Designer, “return-to-libc attack,” *Bugtraq, Aug*, 1997.

[10] H. Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86),” in *Proceedings of CCS 2007*, S. De Capitani di Vimercati and P. Syverson, Eds. ACM Press, Oct. 2007, pp. 552–61.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code-reuse attack,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[13] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[14] T. H. Dang, P. Maniatis, and D. Wagner, “The performance cost of shadow stacks and stack canaries,” in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 555–566.

[15] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-flow bending: On the effectiveness of control-flow integrity.” in *24th USENIX Security Symposium*, 2015.

[16] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *Security and Privacy (SP), 2016 IEEE Symposium on*, 2016.

[17] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, “Block oriented programming: Automating data-only attacks,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1868–1882.

[18] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[19] PAX-TEAM, “Pax aslr (address space layout randomization),” http://pax.grsecurity.net/docs/aslr.txt, 2003.

[20] T. Durden, “Bypassing PaX ASLR protection,” *Phrack magazine #59*, 2002.

[21] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, “Aslr-guard: Stopping address space leakage for code reuse attacks,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 280–291.

[22] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, “Dynamic analysis of malicious code,” *Journal in Computer Virology*, 2006.

[23] D. Wagner and D. Dean, “Intrusion detection via static analysis,” *IEEE Symposium on Security and Privacy*, 2001.

[24] D. Devi and S. Nandi, “Pe file features in detection of packed executables,” *International Journal of Computer Theory and Engineering*, 2012.

[25] M. I. Sharif, V. Yegneswaran, H. Saidi, P. A. Porras, and W. Lee, “Eureka: A framework for enabling static malware analysis,” *ESORICS*, 2008.

[26] I. You and K. Yim, “Malware obfuscation techniques: A brief survey,” *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, 2010.

[27] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” *ACM Comput. Surv.*, 2012.

[28] W. Lee, S. J. Stolfo, and P. K. Chan, “Learning patterns from unix process execution traces for intrusion detection,” *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[29] S. A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion detection using sequences of system calls,” *Journal of Computer Security*, 1998.

[30] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. yong Zhou, and X. Wang, “Effective and efficient malware detection at the end host,” *USENIX Security Symposium*, 2009.

[31] X. Hu, T. cker Chiueh, and K. G. Shin, “Large-scale malware indexing using function-call graphs,” *ACM Conference on Computer and Communications Security*, 2009.

[32] C. Warrender, S. Forrest, and B. A. Pearlmutter, “Detecting intrusions using system calls: Alternative data models,” *IEEE Symposium on Security and Privacy*, 1999.

[33] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu, “Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis,” *DSN*, 2015.

[34] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, “Learning and classification of malware behavior,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 108–125.

[35] J. Z. Kolter and M. A. Maloof, “Learning to detect and classify malicious executables in the wild,” *ournal of Machine Learning Research*, 2006.

[36] J. Aycock, R. deGraaf, and M. J. Jr., “Anti-disassembly using cryptographic hash functions,” *Journal in Computer Virology*, 2006.

[37] K. K. Ispoglou and M. Payer, “malwash: Washing malware to evade dynamic analysis.” in *WOOT*, 2016.

[38] P. Ferrie, “Attacks on virtual machine emulators,” *Symantec Security Response*, 2006.

[39] R. R. Branco, G. N. Barbosa, and P. D. Neto, “Scientific but not academical overview of malware anti-debugging, anti-disassembly and antivm technologies.” [Online]. Available: http://research.dissect.pe/docs/blackhat2012-paper.pdf

[40] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, “Fuzzgen: Automatic fuzzer generation,” in *29th USENIX Security Symposium*. USENIX Association, 2020.

[41] K. Serebryany, “Oss-fuzz - google’s continuous fuzzing service for open source software,” https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany, 2017.

[42] ——, “Oss-fuzz,” https://github.com/google/oss-fuzz.

[43] P. Godefroid, “From blackbox fuzzing to whitebox fuzzing towards verification,” in *Presentation at the 2010 International Symposium on Software Testing and Analysis*, 2010.

[44] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, “Fuzzing: the state of the art,” DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), Tech. Rep., 2012.

[45] M. Zalewski, “American fuzzy lop,” 2015.

[46] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” *IEEE Transactions on Software Engineering*, 2017.

[47] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[48] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, “Collafl: Path sensitive fuzzing,” in *CollAFL: Path Sensitive Fuzzing*, 2018.

[49] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution.” in *NDSS*, 2016.

[50] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[51] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “Qsym: a practical concolic execution engine tailored for hybrid fuzzing,” in *27th USENIX Security Symposium*, 2018.

[52] P. Chen and H. Chen, “Angora: Efficient fuzzing by principled search,” *arXiv preprint arXiv:1803.01307*, 2018.

[53] K. Serebryany, “libfuzzer: A library for coverage-guided fuzz testing (within llvm).”

[54] O. H. Alliance, “Android open source project,” 2011.

[55] “CVE-2019-2176: Heap buffer overflow in libhevcdec,” 2019.

[56] “CVE-2019-2108: libhevc: Stack-buffer-overflow in ihevcd_ref_list,” 2019.

[57] “CVE-2019-2107: libhevc: Multiple heap-buffer overflows in ihevcd_decode,” https://nvd.nist.gov/vuln/detail/CVE-2019-2107, 2019.

[58] “CVE-2019-2106: libhevc: Stack-buffer-underflow in ihevc_sao_edge_offset_class2_chroma_ssse3,” https://nvd.nist.gov/vuln/detail/CVE-2019-2106, 2019.

[59] "CVE-2017-13187: libhevc: An information disclosure vulnerability in the android media framework," https://nvd.nist.gov/vuln/detail/CVE-2017-13187, 2017.

[60] "CVE-2017-0858: libavc: Another vulnerability in the android media framework," https://nvd.nist.gov/vuln/detail/CVE-2017-0858, 2017.

[61] J. Drake, "Stagefright: Scary code in the heart of android," *BlackHat USA*, 2015.

[62] J. Röning, M. Lasko, A. Takanen, and R. Kaksonen, "Protos-systematic approach to eliminate software vulnerabilities," *Invited presentation at Microsoft Research*, 2002.

[63] D. Aitel, "An introduction to spike, the fuzzer creation kit," *presentation slides), Aug*, 2002.

[64] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, 2011.

[65] R. Swiecki, "Honggfuzz," *Available online a t: http://code. google. com/p/honggfuzz*, 2016.

[66] S. Hocevar, "zzufâĂŤmulti-purpose fuzzer," 2011.

[67] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[68] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[69] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[70] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 975–985.

[71] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *International Conference on Automated Software Engineering*, 2009.

[72] ——, "Leveraging test generation and specification mining for automated bug detection without false positives," in *International Conference on Software Engineering*, 2012.

[73] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing API Usages through Semantic Cross-checking," in *25th Usenix Security Symposium*, 2016.

[74] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting ssl usage in applications with sslint," in *2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[75] H. Chen and D. Wagner, "Mops: an infrastructure for examining security properties of software," in *Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[76] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Security and Privacy (SP), 2014 IEEE Symposium on*, 2014.

[77] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2004.

[78] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012.

[79] "The clang development team: Sanitizer coverage," http://clang.llvm.org/docs/SanitizerCoverage.html, 2015.

[80] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[81] G. J. Sullivan, J.-R. Ohm, W.-J. Han, T. Wiegand *et al.*, "Overview of the high efficiency video coding(hevc) standard," *IEEE Transactions on circuits and systems for video technology*, 2012.

[82] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *Usenix Security Symposium*, 1998.

[83] T. Müller, "ASLR smack & laugh reference," *Seminar on Advanced Exploitation Techniques*, 2008.

[84] Kil3r and Bulba, "Bypassing stackguard and stackshield," *Phrack magazine #53*, 2000.

[85] G. Richarte *et al.*, "Four different tricks to bypass stackshield and stackguard protection," *World Wide Web*, 2002.

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

[87] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[88] V. Katoch, "Whitepaper on bypassing aslr/dep," Secfence, Tech. Rep., September 2011.[Online]. Available: http://www.exploit-db.com/wp-content/themes/exploit/docs/17914.pdf, Tech. Rep.

[89] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM." in *23rd USENIX Security Symposium*, 2014.

[90] Microsoft, "Visual studio 2015 — compiler options — enable control flow guard," 2015, https://msdn.microsoft.com/en-us/library/dn919635.aspx.

[91] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, 2018.

[92] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity." in *OSDI*, vol. 14, 2014, p. 00000.

[93] B. Niu and G. Tan, "Modular control-flow integrity," *ACM SIGPLAN Notices*, vol. 49, 2014.

[94] ——, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[95] T. C. Projects, "Control flow integrity," https://www.chromium.org/developers/testing/control-flow-integrity, 2018.

[96] J. Tang and T. M. T. S. Team, "Exploring control flow guard in windows 10," *Available at "http://blog.trendmicro.com/trendlabs-security-intelligence/exploring-control-flow-guard-in-windows-10"*, 2015.

[97] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015.

[98] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[99] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*, 2014.

[100] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection." in *23rd USENIX Security Symposium*, 2014.

[101] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: size does matter in turing-complete return-oriented programming," in *Proceedings of the 6th USENIX conference on Offensive Technologies*. USENIX Association, 2012, pp. 7–7.

[102] J. Salwan and A. Wirth, "ROPGadget," https://github.com/JonathanSalwan/ROPgadget, 2012.

[103] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in *USENIX Security Symposium*, 2011.

[104] A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden, "PSHAPE: Automatically combining gadgets for arbitrary method execution," in *International Workshop on Security and Trust Management*, 2016.

[105] Pakt, "ropc: A turing complete rop compiler," https://github.com/pakt/ropc, 2013.

[106] M. Polychronakis and A. D. Keromytis, "ROP payload detection using speculative code execution," in *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, 2011.

[107] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[108] E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller, "Detecting code reuse attacks with a model of conformant program execution," in *International Symposium on Engineering Secure Software and Systems*, 2014.

[109] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG *et al.*, "ROPecker: A generic and practical approach for defending against ROP attack," 2014.

[110] V. Pappas, "kBouncer: Efficient and transparent rop mitigation," *tech. rep. Citeseer*, 2012.

[111] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses." in *23rd USENIX Security Symposium*, 2014.

[112] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2015.

[113] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," 2017.

[114] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)*, October 2015.

[115] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1675–1689. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134026

[116] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. The MIT press, 2009.

[117] K. Sen, D. Marinov, and G. Agha, “Cute: a concolic unit testing engine for c,” in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[118] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.” in *OSDI*, 2008.

[119] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, “SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis,” in *Security and Privacy (SP), 2016 IEEE Symposium on*, 2016.

[120] A. B. Kahn, “Topological sorting of large networks,” *Communications of the ACM*, 1962.

[121] T. Uno, “Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs,” *Algorithms and Computation*, 1997.

[122] J. Y. Yen, “Finding the k shortest loopless paths in a network,” *management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[123] “CVE-2006-5815: Stack buffer overflow in proftpd 1.3.0,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5815, 2006.

[124] “CVE-2013-2028: Nginx http server chunked encoding buffer overflow 1.4.0,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028, 2013.

[125] “CVE-2012-0809: Format string vulnerability in sudo 1.8.3,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0809, 2012.

[126] “Cve/bug in orzhttpd - format string,” https://www.exploit-db.com/exploits/10282/, 2009.

[127] “CVE-2000-0573: Format string vulnerability in wu-ftpd 2.6.0,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0573, 2001.

[128] “CVE-2002-1496: Heap-based buffer overflow in null http server 0.5.0,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1496, 2004.

[129] “CVE-2001-0144: Integer overflow in openssh 1.2.27,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144, 2001.

[130] “CVE-2014-2299: Buffer overflow in wireshark 1.8.0,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2299, 2014.

[131] “CVE-2006-3747: Off-by-one error in apache 1.3.34,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3747, 2006.

[132] “CVE-2009-1886: Format string vulnerability in smbclient 3.2.12,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1886, 2009.

[133] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, “Automatic generation of data-oriented exploits.” in *24th USENIX Security Symposium*, 2015.

[134] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, 1971, pp. 151–158.

[135] M. V. Yason, “The art of unpacking,” https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf, 2007.

[136] E. Eilam, *Reversing: Secrets of Reverse Engineering*. Wiley; 1 edition, 2005.

[137] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler*. No Starch Press; 2 edition, 2011.

[138] V. Mohan and K. W. Hamlen, “Frankenstein: Stitching malware from benign binaries,” *Usenix WOOT*, 2012.

[139] G. Poulios, C. Ntantogian, and C. Xenakis, “Ropinjector: Using return oriented programming for polymorphism and antivirus evasion,” *Blackhat USA*, 2015.

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

[141] “Metasploit,” https://www.metasploit.com/.

[142] S. Fewer, “Reflective dll injection,” http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf.

[143] P. V. Shijoa and A. Salimb, “Integrated static and dynamic analysis for malware detection,” *ICICT*, 2014.

[144] S. Yusirwan, Y. Prayudi, and I. Riadi, “Implementation of malware analysis using static and dynamic analysis method,” *International Journal of Computer Applications*, 2015.

[145] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[146] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, 1971, pp. 151–158.

[147] U. Müller, “Brainfuck–an eight-instruction turing-complete programming language,” *Available at the Internet address http://en. wikipedia. org/wiki/Brainfuck*, 1993.