# Autonomous Computer Security Game: Techniques, Strategy and Investigation

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical & Computer Engineering

Tiffany Bao

B.S., Computer Science, Peking University

Carnegie Mellon University

Pittsburgh, PA

August, 2018

**Keywords:** Software Security, Cyber Autonomy

*For the trip in 2007.*

# Abstract

Computer security in deployed systems is a dynamic interaction between attackers and defenders. These interactions can be formalized as computer security games between multiple parties, each of which interacts through actions such as finding a zero-day vulnerability, using an exploit, and deploying a patch. Computer security games provide a framework to think through players' choices and consequences, as well as serve as a model of components for optimizing security analysis. In this dissertation, we consider games where each party is modeled as an algorithm. We call these games *autonomous computer security games*.

This dissertation investigates autonomous computer security games from both a game-theoretical and a system perspective. In particular, we study concrete system instances of players as represented by Cyber Reasoning Systems (CRS) found in the DARPA Cyber Grand Challenge (CGC) such as Mayhem and Mechanical Phish. Nonetheless, autonomous computer security games are general games that are also applicable to other scenarios such as cyber warfare.

This dissertation is composed of two main lines of research. First, we research players' strategy based on game-theoretical models. We consider the interaction between multiple players, seek for the optimal strategy corresponding to an equilibrium of the associated game, and explore the factors that affect the outcome of the game. Second, we study critical actions in the theoretical model and investigate the techniques that realize such actions in real systems.

# Acknowledgements

First of all, I would like to thank my advisor Professor David Brumley. You led me into the field of software security, and you have given me unconditional support through my entire Ph.D. You teach me how to be persistent and visionary, and you always stand on my side supporting and advocating my research. I would not have my work done without your constant support and valuable advice.

Next, I would like to thank my committee: Professor Wenke Lee, Professor Giovanni Vigna and Professor Hui Zhang. They gave me insightful suggestions and comments not only for my dissertation but also for my future research and academic career.

Besides my advisor and committee, I have also received great help from Professor David Eckhardt, Professor Anthony Rowe, and Professor Vyas Sekar. I would like to thank them for their patience and kindness. In particular, I would like to thank Ms. Xiaolin Zang for her great care when I was facing challenges in my life.

I would like to acknowledge the funding that I received from DARPA under Contract No. HR00111220009, as well the fellowship from Carnegie Mellon University. Thanks to the financial support, I can fully focus on my research.

I would like to thank all my CMU and UPitt friends: Joseph Brigg, Xiao Chang, Ivan Gotovchits, Mengwen He, Matthew Maurer, Weijing Shi, Maverick Woo, Miao Yu, Tianlong Yu, Yifei Yuan, Carolina Zarate, and everyone else in CyLab. Thank you for being with me and helping me with my research and life. I would also like to thank all my UCSB friends: Yan Shoshitaishvili, Fish Wang, Martina Lindorfer and everyone else in Shellphish. Although Santa Barbara is much

more chilled than I thought, 2016's summer was still a great fun because of you.

In the last but not the least, I would love to thank my family: my father, Dr. Guorui Bao, and my mother, Ms. Xiaoyun You. Thank you for letting me feel loved. There is no word can describe how much you love me. Also, I would love to thank my husband, Dr. Zilin Jiang. I enjoy attending coding competition and learning elliptic curves with you. Meeting you at Carnegie Mellon University makes my Ph.D. from great to perfect.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Software security is critical. According to the Common Vulnerabilities and Exposures (CVE) Entries [112], the number of vulnerabilities reported in 2017 has exceeded 14,000, which was 2.5 times as many as that of in 2016 [3]. Meanwhile, the number and the complexity of software is also growing, and such growth will potentially cause more vulnerabilities in the future [122].

To address security issues in cyberspace and protect the billions of computers running many software, security researchers have been seeking new advanced offense and defense software security techniques. Offense, for example, includes techniques such as automatic exploit generation [19, 37, 65, 66, 99, 101]. Defense, for example, includes techniques such as binary hardening [15, 82, 102, 121] and software patching [86, 109].

However, software security techniques have their merits in practice only to the extent they can be used to achieve a goal. In principle, the goal, under security context, is to maximize one's security outcome. More specifically, different parties — individuals, companies or nations — implement offense and defense techniques as components in networked systems, and the systems strategically interact with each other. Under networked environment, parties need to defend against security attacks and meanwhile may exploit their opponents to achieve their own good purposes. For instance, party A uses its offense component to attack party B for learning B's secret, and party B uses its defense component to protect the secret from being stolen by A.

Currently, the problem of offense and defense strategy for software vulnerabilities remain open. For example, if an individual, a company or a nation discovers a vulnerability, should it disclose or withhold the vulnerability, in order to be secure? It has been shown that disclose right away may not be the best strategy in practice, since patches released along with vulnerability disclosure may induce the generation of the corresponding exploit [34]. If it plans to withhold, then how long should it keep for a secret? These questions need to be answered accurately so that a party will act optimally using the current software security techniques. These questions also need to be answered efficiently. Otherwise, the party may miss the best time to execute the offense or/and defense actions.

For example, the WannaCry attack [9], happening in 2017, is the most serious ransomware attack in history [12]. The attack infected 300,000 computers [11] and caused 4 billion dollars of losses [10]. One of the critical reasons for the severe consequence is the decision made by NSA. NSA first discovered the vulnerability in 2012. Instead of disclosing the vulnerability to the software vendor and have it release the patch, NSA retained the vulnerability for five years from 2012 to 2017. If NSA has had disclosed the vulnerability earlier, the software vendor would have released the patch earlier, and we would have more time to patch the machines before the attack came. Did the NSA's strategy make at least mathematical sense, or was it suboptimal? We will introduce and discuss more details in Section 1.1.

In this dissertation, we *holistically* study software security considering both software security techniques and offense and defense strategy. We abstract the offense and defense interactions in computer security as a *game* involving multiple players, each of which has a set of internal states as well as explicit offense and defense actions. We discuss the case that players are allowed to discover a new vulnerability, create an exploit, generate a patch, decide to play an exploit, decide to release a patch and so on. More specifically, we investigate *autonomous computer security games* and concrete system instances of players as represented by Cyber Reasoning Systems (CRS) found in the DARPA Cyber Grand Challenge (CGC, Section 6.3.1) such as Mayhem [56] and Mechanical Phish [105]. Nonetheless, autonomous computer security games are general games that are also applicable to other scenarios such as cyber warfare. For example, if a government organization such as NSA finds a zero-day vulnerability is discovered, it needs to strategically decide whether it should retain the vulnerability and attack, or disclose the vulnerability and patch. We show that *the autonomous computer security game model (§ 1.3) can be used for calculating optimal strategies (§ 5) and evaluating the strategic impact (§ 6) of binary analysis techniques (§ 3, § 4).*

This dissertation makes contributions to both software security techniques and security game strategy. We propose two new techniques: ByteWeight and ShellSwap, and we develop security games that capture a richer set of actions than previous work. Furthermore, this dissertation uses techniques and strategy to inform what new techniques would help maximize security outcome. For example, we develop ShellSwap, a technique to automatically generate new control flow hijacking exploits with customized shellcodes. This technique improves offense skill; in specific, it gives a party the capacity to *retaliate* by enabling those who have received attacks to generate exploits with their payload. Based on the autonomous computer security game model, we found that if players have the ShellSwap technique, attackers will be less likely to attack. In particular, if all players have the ShellSwap technique and also patch fast, it is possible that no players attack (see Section 6). Overall, we argue that both techniques and strategy are essential and we need to combine them to reason about software vulnerability mitigation.

## 1.1 Real-World Example: The WannaCry Attack

### 1.1.1 The Story

The WannyCra attack [9] is a worldwide cyberattack occurring in May 2017. The attacker used vulnerability CVE-2017-0144 [2] to exploit computers and encrypt the data in compromised machines. After encryption, the attacker asked the victim for ransom payment to decrypt and recover the data in the compromised computer.

The vulnerability CVE-2017-0144 exists in a majority of Windows operating systems, including Microsoft Windows Vista SP2, Windows Server 2008 SP2 and R2 SP1, Windows 7 SP1, Windows 8.1, Windows Server 2012 Gold and R2, Windows RT 8.1, Windows 10 and Windows Server 2016 [2]. Due to the large population of Windows operating system victim users, the WannaCry attack infected 300,000 machines [11] and caused $4 billion losses [10] in the end, and it became a significant ransomware attack in history [12].

We start the study of the WannaCry attack with the timeline of the vulnerability. Figure 1.1 shows the brief timeline of the vulnerability and the WannaCry attack [6]. The detailed information is as follows:

- **2012 (or possibly earlier)**: The vulnerability is identified by the U.S. National Security Agency. The NSA learned that the vulnerability affects many Windows operating systems, and it decided to withhold the vulnerability and produce its exploit called EternalBlue. The EternalBlue exploit was used as a weapon targetting to computers in other countries such as Iran, India, and Syria.
- **January 2017**: The EternalBlue exploit was stolen by the Shadow Broker, and NSA noticed the possible leak of the vulnerability.
- **February 2017**: NSA told Microsoft about the vulnerability.
- **March 2017**: The vulnerability was published as CVE-2017-0144, and the patch was released on March's Patch Tuesday [4].
- **May 2017**: The WannaCry attack started. It was ransomware using vulnerability CVE-2017-0144. The first outbreak lasted for 4 days. Later this attack was attributed to North Korea, according to the U.S. government [8].

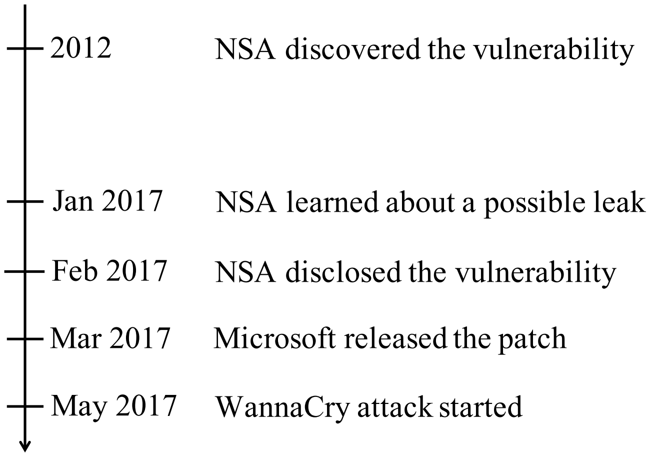

2012
NSA discovered the vulnerability
Jan 2017
NSA learned about a possible leak
Feb 2017
NSA disclosed the vulnerability
Mar 2017
Microsoft released the patch
May 2017
WannaCry attack started

Figure 1.1: The timeline of the WannaCry attack.

## 1.2 Enabling Cyber Autonomy in Software Security

Binary analysis techniques are a group of techniques of program analysis on compiled executables. Binary analysis is capable of investigating specific program behavior such as heap and stack manipulation and the investigation is essential since an exploit is specific to binary executions.

Offense and defense strategy determines the actions that will be executed by each player. In this thesis, players are algorithms that use binary analysis techniques to find vulnerability, patch and create exploits. An optimal strategy considers the best use of automated techniques and maximizes the security interests of a party.

We consider combining both techniques and strategy in this thesis. Moreover, we consider autonomous systems that deal with software programs in discovering vulnerabilities and making offense and defense decisions.

## 1.3 Autonomous Computer Security Game

Autonomous computer security games investigate the attack-defense interaction between multiple players. Players in autonomous computer security games have the capability to discover a vulnerability, create an exploit (to attack) and generate a patch (to defend), and when they discover a vulnerability, they need to choose when to attack and when to defend. Meanwhile, players may not be aware whether or not the other players have already discovered the same vulnerability, so players may not know if they are the first to discover the vulnerability. Under such circumstance, players need to act between defense and attack to maximize its rational outcome. For instance, consider an autonomous computer security game in which each player represents a nation. What should a player act upon discovery of a zero-day vulnerability? Is it better to pass the information to the relevant software vendor and improve everyone's security, or would it be more prudent to keep the vulnerability hidden and develop a zero-day exploit? After developing a zero-day exploit, when should it initiate the attack? Should it attack right away or retain it and use it later? Also, what should a player act when another player attacks it? The above questions will be answered by every player in the game.

We introduce our approach to modeling multi-agent interaction as the *autonomous computer security game*. First, we represent the setup of each agent and its technique components, states and actions. Next, we introduce the interactions between players, and we show how the interactions happen over time by an example. In the end, we list the assumptions of autonomous computer security games.

### 1.3.1 Player Setup

**Technique Components.** In autonomous computer security games, each player is represented as an autonomous system. Although different players have different designs and implementations for the system, they share the common goal, which is to meet its own interests by attacking the other players and protecting themselves. Therefore, we model the player by dividing the autonomous system into components with different functionalities. In specific, we consider three components listed as follows:

- **Vulnerability Awareness Component**

The vulnerability awareness component is responsible for realizing the existence of a vulnerability. When the component acquires a vulnerability, it will trigger the offense and defense components to generate an exploit and a patch for the vulnerability, respectively. In addition, the player will start to decide the strategy for the vulnerability and play the defensive or/and offensive actions.

There are many ways to learn a vulnerability. For example, the component can get a vulnerability by discovering crashes from programs, detecting attacks through network or studying vulnerability databases such as the Common Vulnerabilities and Exposures (CVE) [112] entries. These means can be used simultaneously; one can implement multiple techniques for the vulnerability awareness component, and it will help the autonomous system learn vulnerabilities more effectively and more efficiently.

- **Offense Component**

  The offense component aims to weaponize a vulnerability to exploits that carry out the player's intended execution such as to install backdoors, to exfiltrate sensitive information and so on. A player with the generated exploit can choose to attack the other players, and the attack might bring benefits to the attacker in the autonomous computer security game. This component can be realized by techniques that generates exploits in general, such as automatic exploit generation [19], automatic patch-based generation [34] and ShellSwap [25].

- **Defense Component**

  The defense component is to generate a patch that protects a player from attacks. However, patch releasing takes time, during which a player may still be attacked due to the incomplete process. Patch releasing is also public, in the sense that the released patch will be known by all players in the game. This reflects the reality since patch releasing is usually acted by software vendors, which is a party exclusive from any players and their goal is to patch all users. This component can be realized by automatic patching engine such as

Patcherex [106].

**Player States and Player Actions** For each vulnerability, the players in autonomous computer security games either know it or not know it. We use player state to represent whether or not a player has known the vulnerability. Each player $i$ has a state denoted by $\theta_i$ in each round, where $\theta_i \in \Theta_i = \{\neg\mathrm{D}, \mathrm{D}\}$. $\neg$D refers to the situation in which a player has not yet learned of a vulnerability, while D refers to the situation in which a player knows the vulnerability.

In each round, players choose one of the following actions: {ATTACK, PATCH, NOP, STOCKPILE}, where the semantics of ATTACK, PATCH, and NOP have their literal meaning, and STOCKPILEmeans holding a zero-day for future use.

Players are limited in their actions by their state. In particular, while a player in state $\neg$D can only act NOP, a player in state D can choose an action among ATTACK, PATCH and STOCKPILEbefore the patch is released, and between ATTACK and NOP after the release, depending on their skill at detecting attacks or generating patch-based exploits.

### 1.3.2 Interaction Between Players

The players in autonomous computer security games interact with each other as shown in Figure 1.2. There are two kinds of interactions: attacking and patching. For attacking, it has a culprit and a victim, which we call the attacker and the defender. We assume that if the defender detects the attack, he will know the attacker. For patching, we assume that it is public and thus all players know which player releases the patch.

Interactions happen over time. For example, Figure 1.3 shows the timeline of a game instance of two players (player 1 and player 2). At the beginning of the game, both players are in state of $\neg$D. Player 1 realizes a vulnerability at $T_1$, after which he weaponizes the vulnerability at $T_2$ and generates a patch at $T_4$. After $T_2$ and $T_4$, player 1 starts to attack at $T_3$ and patch at $T_5$. While player 1 is patching, if player 2 has not yet discovered the vulnerability, he will know the vulnerability from the disclosure of the patch, and he will generate exploit and determine when

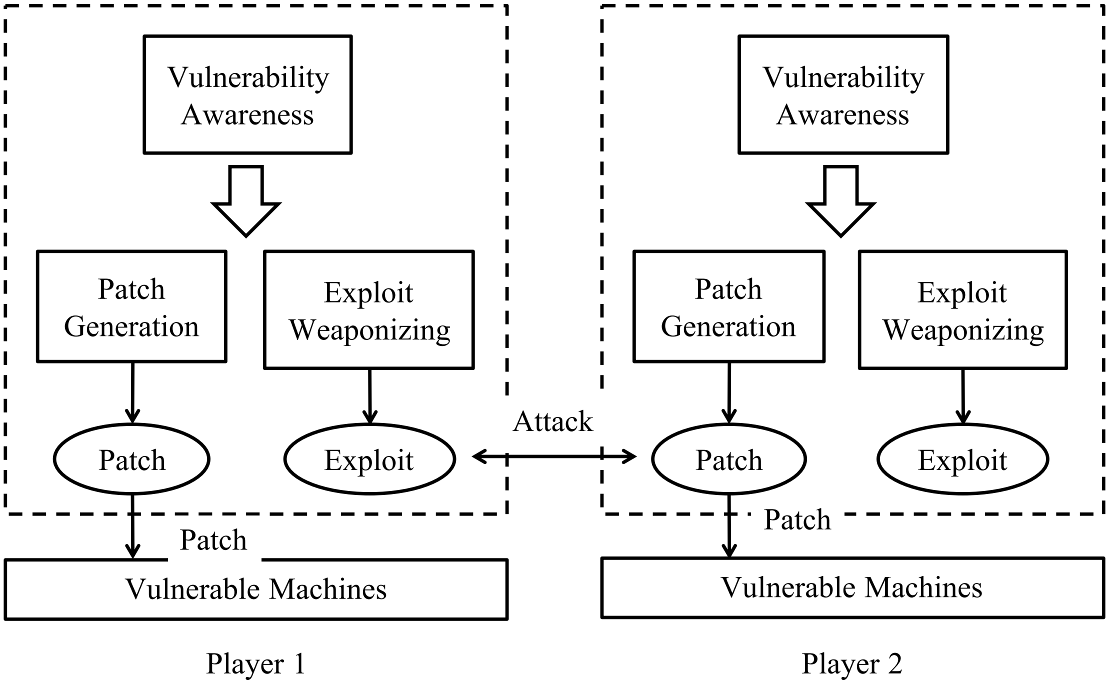
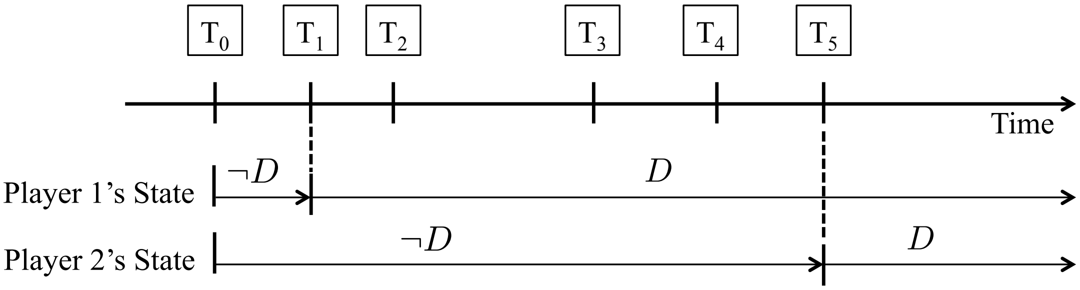


Figure 1.2: The interaction of two autonomous systems (shown underlined).

to attack and when to defend.

**Scope.** In this dissertation, we consider the game within the scope of the following assumptions:

- We assume that players are monitoring their systems, and may probabilistically detect an attack. We also assume they may be able to then *ricochet* the exploit to other players. We note that the detection may come through monitoring the network (in the case of network attacks), or other measures such as honeypots, dynamic analysis of suspicious inputs, etc. For example, Vigilante [41] detects exploits and creates a self-certifying alert (SCA), which is essentially a replayable exploit. We note that such attacks may be detected over a network (e.g., in DEFCON CTFs these are called reflection attacks [95]) or via dynamic analysis, as with Vigilante.
- We assume that patching is always public. Once a patch reveals, players can patch their machines, and those who have not yet generated the exploit can take advantage of the



0. A vulnerability is introduced
1. Player 1 realizes the vulnerability
2. Player 1 weaponizes the vulnerability
3. Player 1 launches an attack
4. Player 1 generates a patch
5. Player 1 starts to patch and Player 2 realizes the vulnerability

Figure 1.3: Timeline for an automatic computer security game

patch information. For example, Brumley et al. shows this can be done in some instances automatically [34].

- We assume that the defender knows the attacker for the detected attacks. We do not consider the situation that a player eavesdrops the communication of the other players to learn a vulnerability from the attacks between the others.
- We focus on one vulnerability for the autonomous computer security model, assuming that vulnerabilities are used independently. Modeling the game with multiple vulnerabilities are challenging since players are uncertain about what vulnerabilities the other players have. We leave this as a task, which is to explore the possible solutions for modeling the autonomous computer security game with multiple vulnerabilities (see Task 4).

## 1.4 Methodology

Techniques and strategy complements in software security, and the research of techniques and strategy interplay with each other. As more techniques are developed, more strategies become

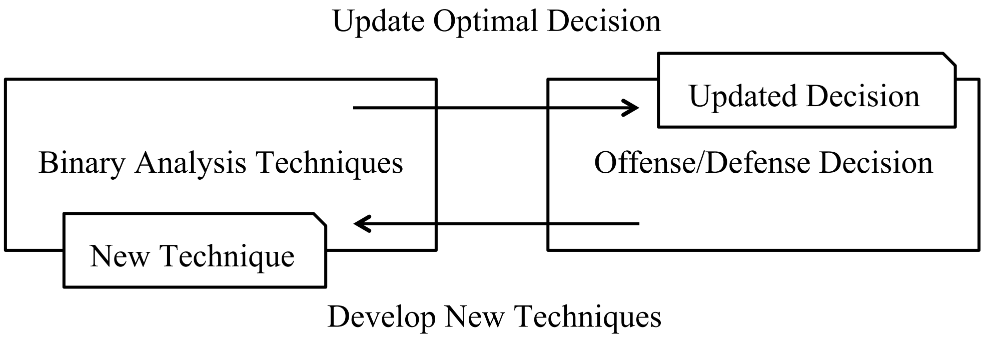


Figure 1.4: The approach to combining techniques and decision making in software security study.

feasible, and thus decision making progress needs to be improved to consider more strategy kinds. On the other hand, as decision making progress highlights the critical techniques for security outcome it guides the development of specific techniques.

Based on the above statement, we form up a methodology of outcome-oriented software security research, shown in Figure 1.4. In this methodology of research, new decision making algorithm is studied as new techniques are proposed, and new techniques are inspired by the investigation of a decision making algorithm.

## 1.5 Contributions

In this dissertation, we model the autonomous computer security game to reason about the mitigation of software vulnerability. We show that the autonomous computer security game model can be used for calculating strategies, identifying the critical components and evaluating the strategic impact of the techniques in autonomous cyber reasoning systems. This dissertation makes the following high-level contributions:

- We provide a framework to systematically reason about vulnerability mitigation, and we propose a holistic approach to software security. Our methodology which combines both techniques and decision making is novel for software security research.
- We build the autonomous computer security game model, which can be used to reason

about decision making for vulnerabilities and qualitatively evaluate known techniques.

- We propose algorithms to find the optimal strategy for vulnerabilities.
- We investigate critical technical components in autonomous cyber reasoning systems and develop those new techniques.

## 1.6 Outline

The dissertation is structured as follows. In Chapter 2 we present the background and the related work of autonomous computer security game. In Chapter 3 and Chapter 4, we introduce two innovative binary techniques, and in Chapter 5, we discuss the algorithm to calculate optimal strategy in the autonomous computer security game model. In Chapter 6, we investigate the autonomous computer security model by showing how to apply it to qualitative technique evaluation, which measures the security impact of a technique or a technique class. In the end, we conclude this dissertation in Chapter 7.

# Chapter 2

# Background

In this chapter, we discuss the background and the related work in software security research. Overall, researchers have worked on both techniques and strategy in software security. However, the current automated binary analysis techniques are often too costly to be scalable, and the strategies are based on models that oversimplify the actual real-world scenarios. Furthermore, there is a disconnect between technique and strategy. Strategies come up with actions, yet strategies overlook the limitations that techniques face when techniques perform actions in practice. For example, a reasonable theoretical strategy for the discovery of a zero-day vulnerability is to patch the vulnerable machines immediately. Unfortunately, it is impossible to instantly create and distribute a patch, so such an action cannot be part of a practical strategy.

## 2.1 Game Models & Nash Equilibrium

We show the relationship of game models in Figure 2.1. In cyber security games, players hide information about their exploits. To accommodate this, we set up the model as an incomplete information game where the players in the model do not know whether other players have discovered a zero-day or not. This assumption is natural: an exploit is only a zero-day from the perspective of each player having never seen it before.

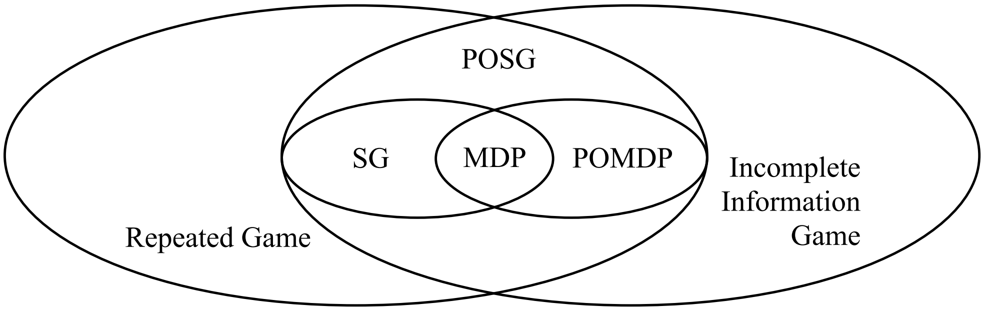


Figure 2.1: The Relationship of game models.

Because cyber security games last from hours to days, it is possible for players to take multiple actions in a game. For example, a player could hold a vulnerability at the beginning of the cyber security game and exploit other players later. In order to support these strategies, we create our model a *multi-stage game*.

The players may find a vulnerability at any time during the game, changing the state of the game from that point. This property is supported by the concept of a *stochastic game* (SG). For an SG, the game played at any given iteration depends probabilistically on the previous round played and the actions of players in the previous round. SG can also be viewed as a multi-player version of Markov Decision Process (MDP) [51].

Combining these concepts, if a game 1) has players with incomplete information, 2) consists of multiple rounds, and 3) players' knowledge may change during the game, then the game is a partially observable, stochastic game (POSG). A POSG can also be considered as a multi-player equivalent of a partially observable Markov decision process (POMDP). A POSG is identical to a POMDP except that instead of a single action, observation, and reward a POSG has one for each player which are expressed together as a joint-action, joint-observation and joint-reward.

A Nash equilibrium is a strategy profile in which players will not have more to gain by changing their strategy. In this dissertation, we will focus on building the game model and finding the Nash equilibrium of the game. Although there are cases of games calling for significantly more refined equilibrium concepts (e.g., perfect Bayesian equilibrium or sequential equilibrium), we follow the argument from previous work [81], claiming that the generally coarser concept of

Nash equilibrium adequately captures the strategic aspects of cyber security games.

## 2.2 Computer Security Game Models

Game theory has been applied in many security contexts, most commonly with a focus on network security or the economics of security, e.g., [18, 36, 39, 57, 77, 78, 80, 83, 94, 111]. We focus exclusively on game theory as it is explicitly applied to the cyber domain. In this subsection, we will describe existing approaches to modeling computer security games and discuss our improvements over these techniques.

Moore et al. [81] proposed the cyber-hawk model in order to find the optimal strategy for both players. The cyber-hawk model describes a game where each player chooses to either keep vulnerabilities private (and create exploits to attack their opponent) or disclose. They conclude that the first player to discover the vulnerability should always attack, precluding vulnerability disclosure.

This model has three limitations. First, it limits the number of players to 2. Second, it assumes that both players will eventually discover the same zero-day vulnerabilities. Third, it models a one-shot game where players are only allowed to make one choice between attack and disclosure. After this choice, the game is over.

The cyber-hawk model raised new questions that need to be explored, such as if a player determines to use a vulnerability offensively, how soon they should commence the attack. Schramm [97] proposes a dynamic model to answer this question. The model indicates that waiting reduces a player's chance of winning the game, which implies that if a player determines to attack, he should act as soon as possible.

The Schramm model relies on a key assumption of *full player awareness*, requiring that players know whether, and how long ago, their opponent discovered a vulnerability. This assumption is unlikely to be valid in real-world scenarios because nations keep the retained vulnerabilities (if they have any at all) secret. Additionally, it is still limited to two players and supports only single taken action, after which the game ends.

We observe three things missing in previous models that are vital for choosing players' best strategies in computer security games. First, players in a computer security game often have multiple actions over multiple rounds. As an example of a multiple round game, consider the NSA case. Although NSA claimed to disclose 91% of all zero-day vulnerabilities, it did not mention whether they first exploited before disclosing or not.

Second, players in a computer security game are uncertain as to the other players' state, specifically, whether other players have discovered vulnerabilities. This uncertainty influences players' decision, as a player must account for all the possible states of the other players in order to maximize his expected utility. Previous approaches cannot be extended to handle multiple steps with partial information and dependencies.

Third, both attacking *and* disclosing reveal the information of a vulnerability. Previous work shows that a patch may be utilized by attackers to generate new exploits [34]. However, we show that attacking leaks information, and we introduce the notion of ricochet into the game theory model. In the automated patch-based exploit generation (APEG) [34] technique, a player infers the vulnerable program point from analyzing a patch and then creates an exploit. Also, in the ricochet attack technique, a player detects an exploit (e.g., through network monitoring or dynamic analysis) and then turns around and uses the exploit against other players. For instance, Costa et al. have proposed monitoring individual programs to detect exploits, and then replaying them as part of their technique for self-certifying filters [41], where the filters self-certify by essentially including a version of the original exploit for replay. Both inadvertent disclosures through attack and patching create new game actions, but previous work does not take it into account. Policy makers and other users of previous models [81, 97] can reach incorrect conclusions and ultimately choose suboptimal strategies.

## 2.3 Automatic Exploit Generation

The technique of automatically generating an exploit with a piece of shellcode is called automatic exploit generation (AEG) [19, 34, 74, 99]. There are two steps for automatic exploit generation: identify the exploitable crash and generate an exploit with the shellcode. For the first step, there are fuzzing, symbolic execution and hybrid methods to find the exploitable crashes. For the second step, Helaan et al. [63] proposed how to place shellcode in memory: scan through the memory and find symbolic memory gaps that are big enough to hold the entire piece of shellcode. For each gap, they try to put shellcode at different offsets by constraining symbolic memory bytes beginning at that offset to the actual bytes of the shellcode. This procedure continues until the shellcode is put in a memory gap or all gaps have been tried.

## 2.4 Automated Patch-based Exploit Generation

Brumley et al. [34] have shown that automated patch-based exploit generation is possible and the technique can be used for generating exploits for real-world vulnerabilities. The existence of such technique highlights that an autonomous system is capable of generating exploits from a released patch, and generating exploits from a released patch is more effective than discovering vulnerabilities from a program. Therefore, we should take into account such technique for the autonomous computer security game.

# Part I

# Binary Analysis Techniques

In this part, we present two new binary analysis techniques that we have developed. These techniques improve the defense and offense in software security. ByteWeight (Chapter 3) is a reverse engineering technique for identifying functions in stripped binaries. It helps to abstract binary programs and can be used for binary patching. ShellSwap (Chapter 4), on the other hand, is a technique to generate new exploits based on a receiving exploit automatically. It helps victims to retaliate if the victims capture the incoming attack.

# Chapter 3

# Function Identification for Stripped Binaries

Function identification is a preliminary and necessary step in many binary analysis techniques and applications. For example, one property of CFI is to constrain inter-function control flow to valid paths. In order to reason about such paths, however, binary-only CFI infrastructures need to be able to identify functions accurately. In particular, COTS-CFI [121], CCFIR [120], MoCFI [50], Abadi et al. [16], and extensions like XFI [54] all depend on accurate function identification to be effective.

CFI is not the only consumer of binary-level function identification techniques. For example, Rendezvous [69] is a search engine that operates at the granularity of function binaries; incorrect function identification can therefore result in incomplete or even incorrect search results. Decompilers such as Phoenix [98], Boomerang [115], and Hex-Rays [61] recover high-level source code from binary code. Naturally, decompilation occurs on only those functions that have been identified in the input binary.

There is disagreement in the community on the efficiency of previous technique. Kruegel et al. argued in 2004 that function start identification can be solved “very well” [72, §4.1] in regular binaries and even some obfuscated ones. Perkins et al. described static function start identification as "a complex task in a stripped x86 executable" [86, §2.2.3] and therefore applied a dynamic approach in their ClearView system. A similar opinion is also shared by Zhang et al., who stated that "it is difficult to identify all function boundaries" [121, §3.2] and used a set of heuristics for this task.

So how good are the current tools at identifying functions from stripped, non-malicious binaries? To find out, we collected a dataset of 2,200 Linux and Windows binaries generated by GNU `gcc`, Intel `icc`, and Microsoft Visual Studio (`VS`) with multiple optimization levels. We then used our dataset to evaluate the most recent release of three popular off-the-shelf solutions for function identification: (i) IDA (v6.5 at submission), used in CodeSurfer/x86 [22], Choi et al.'s work on statically determining binary similarity [38], BinDiff [28], and BinNavi [29]; (ii) the CMU Binary Analysis Platform (BAP-legacy), used in the Phoenix decompiler [98] and the vulnerability analysis tool Mayhem [37]; and (iii) the `unstrip` utility in Dyninst (dated 2012-11-30), used in BinSlayer [31], Sharif et al.'s work on dynamic malware analysis [104], and Sidiroglou et al.'s work on software recovery navigation [108].

Our finding was that while IDA performed better than BAP-legacy and Dyninst on our dataset, its result can still be quite alarming—in our experiment, IDA returned 41.81% true positive rate, 21.38% false negative rate, and 36.81% false positive rate. While there is no doubt that such failures can have a negative impact on downstream security analyses, a real issue is in setting the right expectation on the subject within the security research community. If there is a publicly-available function identification solution where both its mechanism and limitations are well-understood by researchers, then researchers may be come up with creative strategies to cope with the limitations in their own projects. The goal is to explain our process of developing such a solution and to establish its quality through evaluating it against the aforementioned solutions.

We draw inspirations from how BAP-legacy and Dyninst perform function identification since their source code is available. Both solutions rely on fixed, manually-created signatures. Dyninst, at the version we tested, uses the byte signature `0x55` (`push %ebp` in assembly) to recognize function starts in ELF x86 binaries [53]. BAP-legacy v0.7 uses a more complex signature, but it is also manually generated. Unfortunately, the process of manually generating such signatures does not scale well. For example, each new compiler release may introduce new idioms that require new signatures to capture. The myriad of different optimization settings, such as omit frame pointers, may also demand even more signatures. Clearly, we cannot expect to manually catch up.

One approach to recognizing functions is to automatically learn key features and patterns. For example, seminal work by Rosenblum et al. proposed binary function start identification as a supervised machine learning classification problem [1]. They model function start identification as a Conditional Random Field (CRF) in which binary offsets and a number of selected idioms (patterns) appear in the CRF. Since standard inference methods for CRF on large, highly-connected graphs are expensive, Rosenblum et al. adopted feature selection and approximate inference to speed up their model. However, using hardware available in 2008, they needed 150 compute-days just for the feature selection phase on 1,171 binaries.

We propose a new automated analysis for inferring functions and implement it in our ByteWeight system. A key aspect of ByteWeight is the ability to learn signatures for new compilers and optimizations at least one order of magnitude faster than as reported by Rosenblum et al. [1], even after generously accounting for CPU speed increase since 2008. In particular, we avoid using CRFs and feature selection, and instead opt for a simpler model based on learning prefix trees. Our simpler model is scalable using current computing hardware: we finish training 2,064 binaries in under 587 compute-hours. ByteWeight also does not require compiler information of testing binaries, which makes the tool more powerful in practice. In the interest of open science, we also make our tools and datasets available to seed future improvements.

At a high level, we learn signatures for function starts using a weighted prefix tree, and recognize function starts by matching binary fragments with the signatures. Each node in the tree corresponds to either a byte or an instruction, with the path from the root node to any given node representing a possible sequence of bytes or instructions. The weights, which can be learned with a single linear pass over the data set, express the confidence that a sequence of bytes or instructions corresponds to a function start. After function start identification, we then use value set analysis (VSA) [22] with an incremental control flow recovery algorithm to find function bodies with instructions, and extract function boundaries.

To evaluate our techniques, we perform a large-scale experiment and provide empirical numbers on how well these tools work in practice. Based on 2,200 binaries across operating systems, compilers and optimization options, our results show that ByteWeight has a precision and recall of 97.30% and 97.44% respectively for function *start* identification. ByteWeight also has a precision and recall of 92.84% and 92.96% for function *boundary* identification. Our tool is adaptive for varying compilers and therefore more general than current pattern matching methods.

In general, we make the following contributions for solving the function identification problem:

- We propose a new function start identification algorithm based on prefix trees. Our approach is automatic and does not require a priori compiler information (see §3.3). Our approach models the function start identification problem in a novel way that makes it amenable to much faster learning algorithms.
- We evaluate our method on a large test suite across operating systems, compilers, and compiling optimizations. Our model achieves better accuracy than previously available tools.
- We make our test infrastructure, data set, implementation, and results public in an effort to promote open science (see §4.5).

## 3.1 Function Identification Example

We start with a simple example written in C, shown in Code 3.1. In this program, three functions are stored as function pointers in the array `funcs`. When the program is run, input from the user dictates which function gets called, as well as the function arguments. We compiled this

```c
#include <stdio.h>
#include <string.h>
#define MAX 10
void sum(char *a, char *b)
{
    printf("%s + %s = %d\n", a, b, atoi(a) + atoi(b));
}
void sub(char *a, char *b)
{
    printf("%s - %s = %d\n", a, b, atoi(a) - atoi(b));
}
void assign(char *a, char *b)
{
    char pre_b[MAX];
    strcpy(pre_b, b);
    strcpy(b, a);
    printf("b is changed from %s to %s\n", pre_b, b);
}
int main(int argc, char **argv)
{
    void (*funcs[3])(char *x, char *y);
    int f;
    char a[MAX], b[MAX];
    funcs[0] = sum;
    funcs[1] = sub;
    funcs[2] = assign;
    scanf("%d %s %s", &f, a, b);
    (*funcs[f])(a, b);
    return 0;
}
```

Code 3.1: Function identification example in C Code. IDA 6.5 failed to identify functions `sum`, `sub`, and `assign` in the compiled binary with source code shown above.

example code on Linux Debian 7.2 x86-64 using `gcc` with `-O3`, and stripped the binary using the command `strip`. The disassembly of the compiled code is shown in Code 3.2. We then used IDA to disassemble the binary and perform function identification. Many security tools use IDA in this way as a first step before performing additional analysis [35, 68, 85]. Unfortunately, for our example program IDA failed to identify the functions `sum`, `sub`, and `assign`.

IDA's failure to identify these three critical functions has significant implications for security analyses that rely on accurate function boundary identification. Recall that the CFI security policy dictates that runtime execution must follow a path of the static control flow graph (CFG). In

```
00400660 <assign>:
mov     %rbx,-0x10(%rsp)
mov     %rbp,-0x8(%rsp)
sub     $0x28,%rsp
mov     %rdi,%rbp
lea     0xf(%rsp),%rdi
...

004006b0 <sub>:
mov     %rbx,-0x18(%rsp)
mov     %rbp,-0x10(%rsp)
mov     %rsi,%rbx
mov     %r12,-0x8(%rsp)
xor     %eax,%eax
sub     $0x18,%rsp
...

00400710 <sum>:
mov     %rbx,-0x18(%rsp)
mov     %rbp,-0x10(%rsp)
mov     %rsi,%rbx
mov     %r12,-0x8(%rsp)
xor     %eax,%eax
sub     $0x18,%rsp
...
```

Code 3.2: Assembly compiled by `gcc -O3`.

this case, when the CFG is recovered by first identifying functions using IDA, any call to `sum`, `sub`, or `assign` would be incorrectly disallowed, breaking legitimate program behavior. Indeed, any indirect jump to an unidentified or mis-identified function will be blocked by CFI. The greater the number of functions missed, the more legitimate software functionality incorrectly lost. Secondly, suppose we are checking code for potential security-critical bugs. In our sample program, the function `assign` is vulnerable to a buffer overflow attack, but is not identified by IDA as a function. For tools like ClearView [86] that operate on binaries at the function level, missing functions can mean missing vulnerabilities.

## 3.2 Problem Definition and Challenges

The goal of function identification is to determine the set of functions in a binary. Accuracy is compared to a special debug build with ground truth. Determining what functions exist and which bytes belong to which functions is trivial if debug information is present. For example, “unstripped” Linux binaries contain a symbol table that maps function names to locations in a binary, and Microsoft program database (PDB) information contains similar information for Windows binaries. We start with notation to make our problem definition precise and then formally define three function identification problems. We then describe several challenges to any approach or algorithm that addresses the function identification problems. In subsequent sections we provide our approach.

### 3.2.1 Notation and Definitions

A binary program is divided into a number of sections. Each section is given a type, such as code, data, read-only data, and so on. In this dissertation, we only consider executable code, which we treat as a binary string.

Let $B$ denote a binary string. For concreteness, think of this as a binary string from the `.text` section in a Linux executable. Let $B[i]$ denote the $i^{th}$ byte of a binary string, and $B[i : i+j]$ refer to the list of contiguous bytes $B[i], B[i+1], \dots, B[i+j-1]$. Thus, $B[i : i+j]$ is $j$-bytes long (with $j \geq 0$).

Each byte in an executable is associated with an *address*. The address of byte $i$ is calculated with respect to a fixed section offset, i.e., if the section offset is $\omega$, the address of byte $i$ is $i + \omega$. For convenience, we omit the offset, and refer to $i$ as the $i^{th}$ address. Since the real address can always be calculated by adding the fixed offset, this can be done without loss of generality.

A function $F_i$ in a binary $B$ is a list of addresses corresponding to statements in either a function from the original compiled language or a function introduced directly by the compiler, denoted as

$$F = \{B[i], B[j], \dots, B[k]\}$$

Note that function bytes need not be a set of contiguous addresses. We elaborate in §4.1.2 on real optimizations that result in high-level functions being compiled to a set of non-contiguous intervals of instructions.

Towards our goal of determining which bytes of a binary belong to which functions, we define the *set of functions* in a binary

$$\mathbf{FUNCS}(B) = \{F_1, F_2, \ldots, F_k\}.$$

Note that functions may share bytes, i.e., it may be that $F_1 \cap F_2 \neq \emptyset$. We give examples in §4.1.2 where this is the case.

We call the lowest address of a function $F_i$ the *function start* address $s_i$, i.e., $s_i = \min(F_i)$. The *function end* address $e_i$ is the maximum byte in a function body, i.e., $e_i = \max(F_i)$. We define the *function boundary* $(s_i, e_i)$ as the function start and end addresses for $F_i$.

In order to evaluate function identification algorithms, we define ground truth in terms of oracles, which may have a number of implementations:

**Function Oracle.** $\mathbf{O}_{\text{func}}$ is an oracle that, given a binary $B$, returns a list of functions $\mathbf{FUNCS}(B)$ where each $F_i$ is a set of bytes representing higher-level function $i$, as defined above.

**Boundary Oracle.** $\mathbf{O}_{\text{bound}}$ is an oracle that, given $B$, returns the set of function boundaries $\{(s_1, e_1), (s_2, e_2), \ldots, (s_k, e_k)\}$.

**Start Oracle.** $\mathbf{O}_{\text{start}}$ is an oracle that, given $B$, returns the set of function start addresses $\{s_1, s_2, \ldots, s_k\}$.

These oracles are successively less powerful. For example, implementing a boundary oracle $\mathbf{O}_{\text{bound}}$ from a function oracle $\mathbf{O}_{\text{func}}$ requires simply taking the minimum and maximum element of each $F_i$. Similarly, a start oracle $\mathbf{O}_{\text{start}}$ can be implemented from either $\mathbf{O}_{\text{func}}$ or $\mathbf{O}_{\text{bound}}$ by finding the minimum element of each $F_i$.

We do not restrict ourselves to a specific oracle implementation, as realizable oracles may vary across operating system and compiler. For example, the boundary oracle can be implemented by retaining debug information for Windows or Linux binaries. The function oracle can be implemented by instrumenting a compiler to output a list of instruction addresses included in each compiled function.

### 3.2.2 Problem Definition

With the above definitions, we are now ready to state our problem definitions. We start with the least powerful identification (function start) and build up to the most difficult one (entire function).

**Definition 3.2.1.** The *Function Start Identification* (FSI) problem is to output the complete list of function starts $\{s_1, s_2, \ldots, s_k\}$ given a binary $B$ compiled from a source with $k$ functions.

Suppose there is an algorithm $\mathcal{A}_{\mathsf{FSI}}(B)$ for the FSI problem which outputs $S = \{s_1, s_2, \ldots, s_k\}$. Then:

- The set of true positives, TP, is $S \cap \mathbf{O}_{\text{start}}(B)$.
- The set of false positives, FP, is $S - \mathbf{O}_{\text{start}}(B)$.
- The set of false negatives, FN, is $\mathbf{O}_{\text{start}}(B) - S$.

We also define precision and recall. Roughly speaking, precision reflects the number of times an identified function start is really a function start. A high precision means that most identified functions are indeed functions, whereas a low precision means that some sequences are incorrectly identified as functions. Recall is the measurement describing how many functions were identified within a binary. A high recall means an algorithm detected most functions, whereas a low recall means most functions were missed. Mathematically, they can be expressed as

$$\text{Precision} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FP}|}$$

and

$$\text{Recall} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FN}|}.$$

A more difficult problem is to identify both the start *and* end addresses for a function:

**Definition 3.2.2.** The *Function Boundary Identification* (FBI) problem is to identify the start and end bytes $(s_i, e_i)$ for each function $i$ in a binary, i.e., $S = \{(s_1, e_1), (s_2, e_2), \ldots, (s_k, e_k)\}$, given a binary $B$ compiled from a source with $k$ identified functions.

Suppose there is an algorithm $\mathcal{A}_{\mathsf{FBI}}(B)$ for the FBI problem which outputs

$$S = \{(s_1, e_1), (s_2, e_2), \ldots, (s_k, e_k)\}$$

We then define true positives, false positives, and false negatives similarly to above with the additional requirement that *both* the start and end addresses must match the output of the boundary oracle, i.e., for oracle output $(s_{gt}, e_{gt})$ and algorithm output $(s_{\mathcal{A}}, e_{\mathcal{A}})$, a positive match requires $s_{gt} = s_{\mathcal{A}}$ *and* $e_{gt} = e_{\mathcal{A}}$. A false negative occurs if either the start or end address is wrong. Precision and recall are defined analogously to the FSI problem.

Finally, we define the general function identification problem:

**Definition 3.2.3.** The *Function Identification* (FI) problem is to output a set $\{F_1, F_2, \ldots, F_k\}$ where each $F_i$ is a list of bytes corresponding to high-level function $i$ given a binary $B$ with $k$ identified functions.

We define true positives, false positives, false negatives, precision, and recall for the FI problem in the same ways as FSI and FBI but add the requirement that all bytes of a function must be matched between agorithm and oracle.

The above problem definitions form a natural hierarchy, where function start identification is the easiest and full function identification is the most difficult. For example, an algorithm $\mathcal{A}_{\mathsf{FBI}}$ for function boundaries can solve the function start problem by returning the start element of each tuple. Similarly, an algorithm for the function identification problem needs only return the maximum and minimum element to solve the function boundary identification problem.

### 3.2.3 Challenges

Identifying functions in binary code is made difficult by optimizing compilers, which can manipulate functions in unexpected ways. In this section we highlight several challenges posed by the behavior of optimizing compilers.

**Not every byte belongs to a function.** Compilers may introduce extra instructions for alignment and padding between or within a function. This means that not every instruction or byte must belong to a function. For example, suppose we have symbol table information for a binary $B$. One naive algorithm is to first sort symbol-table entries by address, and then ascribe each byte between entry $f_i$ and $f_{i+1}$ as belonging to function $f_i$. This algorithm has appeared in several binary analysis platforms used in security research, such as versions of BAP-legacy [26] and BitBlaze [30]. This heuristic is flawed, however. For example, in Code 3.3, lines 7–8 are not owned by any function.

```
 1  <func1>:
 2  100000e20:   push   %rbp
 3  100000e21:   mov    %rsp,%rbp
 4  100000e24:   lea    0x69(%rip),%rdi
 5  100000e2b:   pop    %rbp
 6  100000e2c:   jmpq   100000e5e <_puts$stub>
 7  100000e31:   nopl   0x0(%rax)
 8  100000e38:   nopl   0x0(%rax,%rax,1)
 9  <func2>:
10  ...
```

Code 3.3: Lines 7–8 do not belong to any function.

**Functions may be non-contiguous.** Functions may have gaps. The gaps can be jump tables, data, or even instructions for completely different functions [93]. As noted by Harris and Miller [62], function sharing code can also lead to non-contiguous functions. Code 3.4 shows code that starts out with the function `ConvertDefaultLocale`. Midway through the function at lines 17–21, however, the compiler decided to include a few lines of code for

`FindNextFileW` as an optimization. Many binary analysis platforms, such as BAP-legacy [26] and BitBlaze [30], are not able to handle non-contiguous functions.

```
 1  <ConvertDefaultLocale>:
 2  7c8383ff:   mov     %edi,%edi
 3  7c838401:   push    %ebp
 4  ...
 5  7c83840c:   jz      7c848556
 6  7c838412:   test    %eax, %eax
 7  7c838414:   jz      7c83965c
 8  7c83841a:   mov     $1024,%ecx
 9  7c83841f:   cmp     %ecx,%eax
10  7c838421:   jz      7c83965c
11  7c838427:   test    $252,%ah
12  7c83842a:   jnz     7c838442
13  7c83842c:   mov     %eax,%edx
14  ...
15  7c838442:   pop     %ebp
16  7c838443:   ret     4
17  ; Chunk of function <FindNextFileW>
18  7c838446:   push    6
19  7c838448:   call    sub_7c80935e
20  7c83844d:
21  ; End of chunk
22  ...
23  7c83965c:   call    <GetUserDefaultLCID>
24  7c890661:   jmp     7c838442
25  ...
26  7c848556:   mov     $8,%eax
27  7c84855b:   jmp     7c838442
```

Code 3.4: Lines 17–21 show code from `FindNextFileW` included in the middle of `ConvertDefaultLocale`.

**Functions may not be reachable.** A function may be dead code and never called, but nonetheless appear in the binary. Recognizing such functions is still important in many security scenarios. For example, suppose two malware samples both contain a unique, identifying, yet uncalled function. Then the two malware samples are likely related even though the function is never called. One consequence of this is that techniques based solely on recursive disassembling from program start are not well-suited to solve the function identification problem. A recursive disassembler only disassembles bytes that occur along some control flow path, and thus by definition will miss functions that are not called.

Unreachability may occur for several reasons, including compiler optimizations. For example, Code 3.5 and Code 3.6 shows a function for computing factorials called `fac`. When compiled by `gcc -O3`, the result of the call to `fac` is precomputed and inlined. Although the code of `fac` appears, it is never called in the binary code. Security policies such as CFI and XFI must be aware of all low-level functions, not just those in the original code.

```
int fac(int x)
{
  if (x == 1) return 1;
  else return x * fac(x - 1);
}

void main(int argc, char **argv)
{
  printf("%d", fac(10));
}
```

Code 3.5: The source program of the unreachable code example.

```
80483f0 <fac>:
...
8048410 <main>:
...
movl    $0x375f00,0x4(%esp)
movl    $0x8048510,(%esp)
call    8048300  ;call printf without fac
xor     %eax,%eax
add     $0x8,%esp
pop     %ebp
ret
```

Code 3.6: The assembly of the unreachable code exmaple, compiled by `gcc -O2`.

**Functions may have multiple entries.** High-level languages use functions as an abstraction with a single entry. When compiled, however, functions may have multiple entries as a result of specialization. For example, the `icc` compiler with `-O1` specialized the `chown_failure_ok` function in GNU LIBC. As shown in Code 3.7 and Code 3.8, a new function entry called `chown_failure_ok.` (note the period) is added for use when invoking `chown_failure_ok` with `NULL`. The compiled binary has both symbol table entries. Unlike shared code for two functions that were originally separate, the compiler here has introduced shared code via multiple entries as an optimization.

Identifying both functions is necessary in many security scenarios, e.g., CFI needs to identify each function entry point for safety, and realize that both are possible targets. More generally, any binary rewriting for protection (e.g., memory safety, control safety, etc.) would need to reason about both entry points.

```
extern bool
chown_failure_ok (struct cp_options const *x)
{
  return ((errno == EPERM || errno == EINVAL)
          && !x->chown_privileges);
}
```

Code 3.7: The source code of the multiple-entry function `chown_failure_ok`.

```
<chown_failure_ok>:
804f544:        mov    0x4(%esp),%eax
<chown_failure_ok.>:
804f548:        push   %esi
804f549:        push   %esi
804f54a:        push   %esi
...
```

Code 3.8: The assembly code of the multiple-entry function `chown_failure_ok`, compiled by `icc -O1`.

**Functions may be removed.** Functions can be removed by function inlining, especially small functions. Compilers perform function-inlining to reduce function call overhead and expose more optimization opportunities. For example, the function `utimens_symlink` is inlined into the function `copy_internal` when compiled by `gcc` with `-O2`. The source code and assembly code are shown in Code 3.9 and Code 3.10. Note that function inlining does not have to be explicitly declared with `inline` annotation in source code. Many compilers inline functions by default unless explicitly disabled with options such as `-fno-deault-inline` [59]. This indicates that for those binary analysis techniques which need function information, even though source code is accessible, a robust function identification technique should still operate on the program binary. If using source code, function identification may be less precise due to functions that are inlined during compilation.

```
static inline int
utimens_symlink (char const *file,
                 struct timespec const *timespec)
{
  int err = lutimens (file, timespec);
  if (err && errno == ENOSYS)
    err = 0;
  return err;
}

static bool
copy_internal (char const *src_name,
               char const *dst_name,
               ...)
{
  ...
  if ((dest_is_symlink
      ?utimens_symlink (dst_name,
                        timespec)
      :utimens (dst_name, timespec))
      != 0)
  ...
}
```

Code 3.9: The source code of a program with function `utimens_symlink` removed due to function inlining optimization.

**Each compilation is different.** Binary code is not only heavily influenced by the compiler but also the compiler version and specific optimizations employed. For example, `icc` does not pre-compute the result of `fac` in Code 3.5, but `gcc` does. Even different versions of a compiler may change code. For example, traditionally `gcc` (e.g., version 3) would only omit the use of the frame pointer register `%ebp` when given the `-fomit-frame-pointer` option. Recent versions of `gcc` (such as version 4.2), however, opportunistically omit the frame pointer when compiled with `-O1` and `-O2`. As a result several tools that identified functions by scanning for

```
<copy_internal>:
100003170:      push   %rbp
100003171:      mov    %rsp,%rbp
100003174:      push   %r15
100003176:      push   %r14
...
10000468c:      test   %r14b,%r14b
10000468f:      je     100005bfd
100004695:      lea    -0x738(%rbp),%rsi
10000469c:      mov    -0x750(%rbp),%rdi
1000046a3:      callq  10000d020 <_lutimens>
1000046a8:      test   %eax,%eax
1000046aa:      mov    %eax,%ebx
...
```

Code 3.10: The assembly code of a program with function `utimens_symlink` removed due to function inlining optimization, compiled by `gcc -O2`.

`push %ebp` break. For example, Dyninst, used for instrumentation in several security projects, relies on this heuristic to identify functions and breaks on recent versions of `gcc`.

In conclusion, functions are diversified in binary as a result of compilation and optimization. Function boundary identification, as a fundamental step for binary analysis so that would bring cumulative effect on the advanced techniques based on this, must be tackled comprehensively. A successful function boundary identification tool should be able to overcome these challenges.

## 3.3 ByteWeight

In this section, we detail the design and algorithms used by ByteWeight to solve the function identification problems. We first start with the FSI problem, and then move to the more general function identification problem.

We cast FSI as a machine learning classification problem where the goal is to label each byte of a binary as either a function start or not. We use machine learning to automatically generate literal patterns so that ByteWeight can handle new compilers and new optimizations without relying on manually generated patterns or heuristics. Our algorithm works with both byte sequences and disassembled instruction sequences.

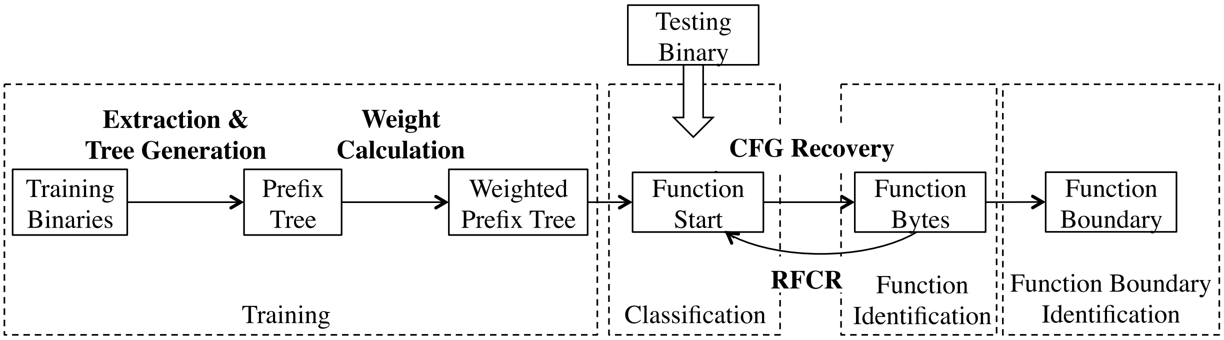


Figure 3.1: The ByteWeight function boundary inference approach.

Our overall system is shown in Figure 3.1. Like any classification problem, we have a training phase followed by a classification phase. During training, we first compile a reference corpus of source code to produce binaries where the start addresses are known. At a high level, our algorithm creates a weighted prefix tree of known function start byte or instruction sequences. We *weight* vertices in the prefix tree by computing the ratio of true positives to the sum of true and false positives for each sequence in the reference data set. We have designed and implemented two variations of ByteWeight: one working with raw bytes and one with normalized disassembled instructions. Both use the same overall algorithm and data structures. We show in our evaluation that the normalization approach provides higher precision and recall, and costs less time (experiment 3.4.2).

In the classification phase, we use the weighted prefix tree to determine whether a given sequence of bytes or instructions corresponds to a function start. We say that a sequence corresponds to a function start if the corresponding *terminal node* in the prefix tree has a weight value larger than the threshold $t$. In the case where the sequence exactly matches a path in the prefix tree, the terminal node is the final node in this path. If the sequence does not exactly match a path in the tree, the terminal node is the last matched node in the sequence.

Once we identify function starts, we infer the remaining bytes (and instructions) that belong to a function using a CFG recovery algorithm. The algorithm incrementally determines the CFG using a variant of VSA [22]. If an indirect jump depends on the value of a register, then we over-approximate a solution to the function identification problem by adding edges that correspond to locations approximated using VSA.

### 3.3.1 Learning Phase

The input to the learning phase is a corpus of training binaries $\mathbb{T}$, and a maximum sequence length $\ell > 0$. $\ell$ serves as a bound on the maximum tree height.

In ByteWeight, we first generate the oracle $\mathbf{O}_{\text{bound}}$ by compiling known source using a variety of optimization levels while retaining debug information. The debug information gives us the required $(s_i, e_i)$ pair for each function $i$ in the binary.

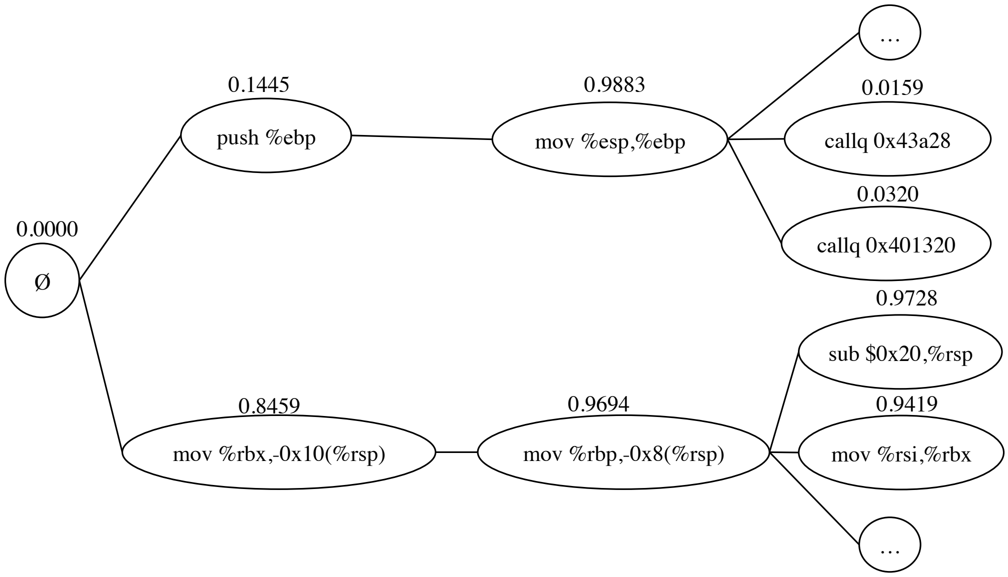
In this dissertation, we consider two possibilities: learning over raw bytes and learning over normalized instructions. We refer to both raw bytes and instructions as a sequence of elements. The sequence length $\ell$ determines how many raw sequential bytes or instructions we consider for training.

**Step 1: Extract first $\ell$ elements for each function (Extraction).** In the first step, we iterate over all $(s_i, e_i)$ pairs and extract the first $\ell$ elements. If there are fewer than $\ell$ elements in the function, we extract the maximum number of elements. For raw bytes, this is $B[s : s + \ell]$ bytes, and for instructions, it is the first $\ell$ instructions disassembled linearly starting from $B[s]$.

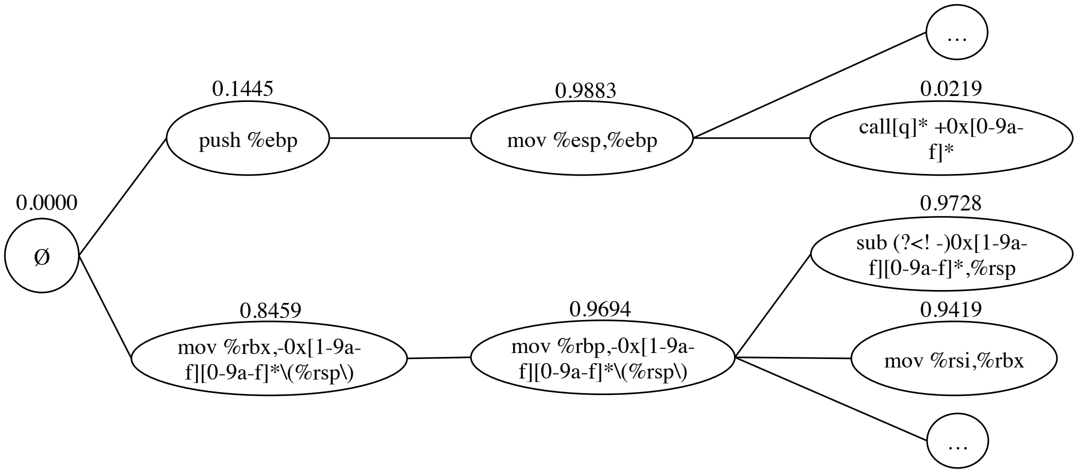
**Step 2: Generate a prefix tree (Tree Generation).** In step 2, we generate a *prefix tree* from the extracted sequences to represent all possible function start sequences up to $\ell$ elements.

A prefix tree, also called a trie, is a data structure enabling efficient information retrieval. In the tree, each non-root node has an associated byte or instruction. The sequence for a node $n$ is represented by the elements that appear on the path from the root to $n$. Note that the tree represents all strings up to $\ell$ elements, not just exactly $\ell$ elements.

Figure 3.2a shows an example tree on instructions, where node `callq 0x43a28` represents the instruction sequence:



(a) Unnormalized


0.0000
Ø
0.1445
push %ebp
0.9883
mov %esp,%ebp
…
0.0219
call[q]* +0x[0-9a-f]*
0.8459
mov %rbx,-0x[1-9a-f][0-9a-f]*\(%rsp\)
0.9694
mov %rbp,-0x[1-9a-f][0-9a-f]*\(%rsp\)
0.9728
sub (?<! -)0x[1-9a-f][0-9a-f]*,%rsp
0.9419
mov %rsi,%rbx
…

(b) Normalized

Figure 3.2: Example of unnormalized (a) and normalized (b) prefix tree. Weight is shown above its corresponding node.

```
push    %ebp              ; saved stack pointer
mov     %esp,%ebp         ; establish new frame
callq   0x43a28           ; call another function
```

If the sequence is over bytes, the prefix tree is calculated directly, although our experiments indicate that a prefix tree calculated over normalized instructions fairs better. We perform two types of normalization: immediate number normalization and call & jump instruction normalization. As shown in Table 3.1, normalization takes an instruction as input and generalizes it so that it can match against very similar, but not identical instructions. These two types of normalization help us improve recall at the cost of a little precision (Table 3.2). In our running example, only the function `assign` is recognized as a function start when matched against the unnormalized prefix tree (Figure 3.2a), while functions `assign`, `sub`, and `sum` can all be recognized when matched against the normalized prefix tree (Figure 3.2b).

| Type | | Unnormalized Signature | Normalized Signature |
|---|---|---|---|
| Immediate | all | `mov $0xaa,%eax` | `mov \$-*0x[0-9a-f]+,%eax` |
| | | `mov %gs:0x0,%eax` | `mov %gs:-*0x[0-9a-f]+,%eax` |
| | | `mov 0x80502c0,%eax` | `mov -*0x[0-9a-f]+,%eax` |
| | zero | `mov $0xaa,%eax` | `mov \$-*0x[1-9a-f][0-9a-f]*,%eax` |
| | | `mov %gs:0x0,%eax` | `mov %gs:0x0+,%eax` |
| | | `mov 0x80502c0,%eax` | `mov -*0x[1-9a-f][0-9a-f]*,%eax` |
| | positive | `mov $0xaa,%eax` | `mov \$(?<! -)0x[1-9a-f][0-9a-f]*,%eax` |
| | | `mov %gs:0x0,%eax` | `mov %gs:-0x[0-9a-f]+\|0x0+,%eax` |
| | | `mov 0x80502c0,%eax` | `mov (?<! -)0x[1-9a-f][0-9a-f]*,%eax` |
| | negative | `mov $0xaa,%eax` | `mov \$(?<! -)0x[0-9a-f]+,%eax` |
| | | `mov %gs:0x0,%eax` | `mov %gs:(?<! -)0x[0-9a-f]+,%eax` |
| | | `mov 0x80502c0,%eax` | `mov (?<! -)0x[0-9a-f]+,%eax` |
| | | `movzwl -0x6c(%ebp),%eax` | `movzl -0x[1-9a-f][0-9a-f]*\(%ebp\),%eax` |
| | npz | `mov $0xaa,%eax` | `mov \$(?<! -)0x[1-9a-f][0-9a-f]*,%eax` |
| | | `mov %gs:0x0,%eax` | `mov %gs:0x0+,%eax` |
| | | `mov 0x80502c0,%eax` | `mov (?<! -)0x[1-9a-f][0-9a-f]*,%eax` |
| | | `movzwl -0x6c(%ebp),%eax` | `movzl -0x[1-9a-f][0-9a-f]*\(%ebp\),%eax` |
| Call & Jump | | `call 0x804cf32` | `call[q]* +0x[0-9a-f]*` |

Table 3.1: Normalizations in signature. For immediate normalization, we generalize immediate operands. There are five kinds of generalization: all, zero, positive, negative, and npz. For jump and call instruction normalization, we generalize callee and jump addresses.

**Step 3: Calculate tree weights (Weight Calculation).** The prefix tree represents possible function start sequences up to $\ell$ elements. For each node, we assign a weight that represents the likelihood that the sequence corresponding to the path from the root node to this node is a function start in the training set. For example, according to Figure 3.2, the weight of node `push %ebp` is 0.1445, which means that during training, 14.45% of all sequences with prefix of `push %ebp` were truly function starts, while 85.55% were not.

To calculate the weight, we first count the number of occurrences $\mathbb{T}_+$ in which each prefix in the tree matches a true function start with respect to the ground truth $\mathbf{O}_{\text{start}}$ for the entire training set $\mathbb{T}$.

Second, we lower the weight of a prefix if it occurs in a binary, but is not a function start. We do this by performing an exhaustive disassembly starting from every address that is *not* a function start [72]. We match each exhaustive disassembly sequence of $\ell$ elements against the tree. We call these *false matches*. The number of false matches $\mathbb{T}_-$ is the number of times a prefix represented in the tree is not a function start in the training set $\mathbb{T}$. The weight for each node $n$ is then the ratio of true positives to overall matches

$$W_n = \frac{\mathbb{T}_+}{\mathbb{T}_+ + \mathbb{T}_-}. \tag{3.1}$$

Since the prefix tree can end up being quite large, it is beneficial to prune the tree of unnecessary nodes. For each node in the tree, we remove all its child nodes if the value of $\mathbb{T}_-$ for this node is 0. For any child node, the value of $\mathbb{T}_-$ is never negative and never larger than the value of $\mathbb{T}_-$ for the parent node. Hence, if $\mathbb{T}_-$ is 0 for a parent node, then the value must be 0 for all of the child nodes as well. The intuition here is that if a child node matches a sequence that is not a function start, then so must the parent node. Thus, if the parent node does not have any false matches, then neither can a child node. Based on Equation 3.1, if $\mathbb{T}_- = 0$ and $\mathbb{T}_+ > 0$, then the weight of the node is 1. Since the child nodes of such a node also have a $\mathbb{T}_-$ value of 0 and are not included in the tree if $\mathbb{T}_+ = 0$, they must also have a weight of 1. As discussed more in Section 3.3.2, child nodes with identical weights are redundant and can safely be removed without affecting classification.

This pruning optimization helps us greatly reduce the space needed by the tree. For example, pruning reduced the number of nodes in the prefix tree from 2,483 to 1,447 for our Windows x86 dataset. Moreover, pruning increases the speed of matching, since we can determine the weight of test sequences after traversing fewer nodes in the tree.

### 3.3.2 Classification Phase Using a Weighted Prefix Tree

The output of the learning phase is a weighted prefix tree (e.g., Figure 3.2). The input to the classification step is a binary $B$, the weighted prefix tree, and a weight threshold $t$.

To classify instructions, we perform exhaustive disassembly of the input binary $B$ and match against the tree. Matching is done by tokenizing the disassembled stream, performing normalization as done during learning, and walking the tree. To classify bytes rather than instructions, we again start at every offset but instead match the raw bytes instead of normalized instructions.

The weight of a sequence is determined by last matching node (the *terminal* node) during the walk. For example, given the tree in Figure 3.2a, and our running example with sequences

```
mov     %rbx,-0x10(%rsp)
mov     %rbp,-0x8(%rsp)
sub     %0x28,%rsp
```

the matching node will be `mov%rbp,-0x8(%rsp)`, giving a weight of 0.9694. However, for another sequence:

```
push    %ebp
and     $0x2,%esp
```

we would have weight 0.1445. We say the sequence is the beginning of a function if the output weight $w$ is not less than the threshold $t$.

### 3.3.3 The Function Identification Problem

At a high level, we address the function identification problem by first determining the start addresses for functions, and then performing static analysis to recover the CFG of instructions that are reachable from the start. Direct control transfers (e.g., direct jumps and calls) are followed using recursive disassembly. Indirect control transfers, e.g., from indirect calls or jump tables, are enumerated using VSA [22]. The final CFG then represents all instructions (and corresponding bytes) that are owned by the function starting at the given address.

CFG recovery starts at a given address and recursively finds new nodes that are connected to found nodes. The process ends when no more vertices are added into graph. Starting at the addresses classified for FSI, CFG recovery recursively adds instructions that are reachable from these starts. A first-in-first-out vertex array is maintained during CFG recovery.

At the beginning, there is only one element  the start address in the array. In each round, we process the first element by exploring new reachable instructions. If the new instruction is not in the array, it will be appended to the end. Elements in the array are handled accordingly until all elements have been processed and no more instructions are added.

If the instruction being processed is a branch mnemonic, the reachable instruction is the branch reference. If it is a call mnemonic, the reachable instructions include both the call reference and the instruction directly following the call instruction. If it is an exit instruction, there will be no new instruction. For the rest of mnemonics, the new instruction is the next one by address. We handle indirect control transfer instruction by VSA: we infer a set that over-approximates the destination of the indirect jump and thus over-approximate the function identification problem.

Note that functions can exit by calling a no-return function such as `exit`. This means that some call instructions in fact never return. To detect these instances, we check the call reference to see if it represents a known no-return function such as `abort` or `exit`.

### 3.3.4 Recursive Function Call Resolution

Pattern matching can miss functions; for example, a function that is written directly in assembly may not obey calling conventions. To catch these kinds of missed functions, we continue to supplement the function start list during CFG recovery. If a call instruction has its callee in the `.text` section, we consider the callee to be a function start. We then do CFG recovery again, starting at the new function start until there are no more functions added into the function start list. We will refer to this strategy as recursive function call resolution (RFCR). In §3.4.3, we discuss the effectiveness of this technique in function start identification.

### 3.3.5 Addressing Challenges

In this section, we describe how ByteWeight addresses the challenges raised in §4.1.2.

First, ByteWeight recovers functions that are unreachable via calls because it does not depend on calls to identify functions. In particular, ByteWeight recovers any function start that matches the learned weighted prefix tree as described above. Similarly, our approach will also learn functions that have multiple entries, provided a similar specialization occurs in the training set. This seems realistic in many scenarios since the number of compiler optimizations that create multiple entry functions are relatively few and can be enumerated during training.

ByteWeight also deals with overlapping byte or instruction sequences provided that there is a unique start address. Consider two functions that start at different addresses, but contain the same bytes. During CFG recovery, ByteWeight will discover that both functions use the same bytes, and attribute the bytes to both functions. ByteWeight can successfully avoid false identification for inlined functions when inlined function does not behave like an empirical function start (does not weighted over threshold in training).

Finally, note that ByteWeight does not need to attribute every byte or instruction to a function. In particular, only bytes (or instructions) that are reachable from the recovered function entries will be owned by a function in the final output.

## 3.4 Evaluation

In this section, we discuss our experiments and performance. ByteWeight is a cross-platform tool which can be run on both Linux and Windows. We used BAP-legacy [26] to construct CFGs. The rest of the implementation consists of 1988 lines of OCaml code and 222 lines of bash shell script. We set up ByteWeight on one desktop machine with a quad-core 3.5GHz i7-3770K CPU and 16GB RAM. Our experiments aimed to address three questions as follows:

- Does ByteWeight's pattern matching model perform better than known models for function start identification? (§3.4.2)
- Does ByteWeight perform function start identification better than existing binary analysis tools? (§3.4.3)
- Does ByteWeight perform function boundary identification better than existing binary analysis tools? (§3.4.4)

In this section, we first describe our data set and ground truth (the oracle), then describe the results of our experiments. We performed three experiments answering the above three questions. In each experiment, we compared ByteWeight against existing tools in terms of both accuracy and speed.

Because ByteWeight needs training, we divided the data into training and testing sets. We used standard 10-fold validation, dividing the element set into 10 sub-sets, applying 1 of the 10 on testing, and using the remaining 9 for training. The overall precision and recall represent the average of each test.

### 3.4.1 Data Set and Ground Truth

Our data set consisted of 2,200 different binaries compiled with four variables:

- **Operating System.** Our evaluation used both Linux and Windows binaries.
- **Instruction Set Architecture (ISA).** Our binaries consisted of both x86 and x86-64 bi-

naries. One reason for varying the ISA is that the calling convention is different, e.g., parameters are passed by default on the stack in Linux on x86, but in registers on x86-64.

- **Compiler.** We used GNU `gcc`, Intel `icc`, and Microsoft `VS`.
- **Optimization Level.** We experimented with the four optimization levels from no optimization to full optimization.

On Linux, our data set consisted of 2,064 binaries in total. The data set contained programs from `coreutils`, `binutils`, and `findutils` compiled with both `gcc` 4.7.2 and `icc` 14.0.1. On Windows, we used `VS` 2010, `VS` 2012, and `VS` 2013 (depending on the requirements of the program) to compile 68 binaries for x86 and x86-64 each. These binaries came from popular open-source projects: putty, 7zip, vim, libsodium, libetpan, HID API, and pbc (a library for protocol buffers). Note that because Microsoft Symbol Server releases only public symbols which do not contain information of private functions, we were unable to use Microsoft Symbol Server for ground truth and include Windows system applications in our experiment.

We obtained ground truth for function boundaries from the symbol table and PDB file for Linux and Windows binaries, respectively. We used `objdump` to parse symbol tables, and `Dia2dump` [52] to parse PDB files. Additionally, we extracted “thunk” addresses from PDB files. While most tools do not take thunks into account, IDA considers thunks in Windows binaries to be special functions. To get a fair result, we filtered out thunks from IDA’s output using the list of thunks extracted from PDB files.

### 3.4.2 Signature Matching Model

Our first experiment evaluated the signature matching model for function start identification. We compared ByteWeight and Rosenblum et al.’s implementation in terms of both accuracy and speed. In order to equally evaluate the signature matching models, recursive function call resolution was not used in this experiment.

The implementation of Rosenblum et al. is available as a matching tool with 12 hard-coded signatures for `gcc` and 41 hard-coded signatures for `icc`. Their learning code was not available, nor was their dataset. Although they evaluated `VS` in their paper, the version of the implementation that we had did not support `VS` and was limited to x86. Each signature has a weight, which is also hard-coded. After calculating the probability for each sequence match, it uses a threshold of 0.5 to filter out function starts. Taking a binary and a compiler name (`gcc` or `icc`), it generates a list of function start addresses. To adapt to their requirements, we divide Linux x86 binaries into two groups by compiler, where each group consists of 516 binaries. We did 10-fold cross validation for ByteWeight, and use the same threshold as Rosenblum et al.'s implementation.

We also evaluated another two varieties of our model: one without normalization, and one with a maximum tree height of 3, which is same as the model used by Rosenblum et al. and ByteWeight (3), respectively.

| | GCC | | | ICC | | |
|---|---|---|---|---|---|---|
| | Precision | Recall | Time(sec) | Precision | Recall | Time(sec) |
| Rosenblum et al. | 0.4909 | 0.4312 | 1172.41 | 0.6080 | 0.6749 | 2178.14 |
| ByteWeight (3) | 0.9103 | 0.8711 | 1417.51 | 0.8948 | 0.8592 | 1905.34 |
| ByteWeight (no-norm) | 0.9877 | 0.9302 | 19994.18 | 0.9727 | 0.9132 | 20894.45 |
| ByteWeight | 0.9726 | 0.9599 | 1468.75 | 0.9725 | 0.9800 | 1927.90 |

Table 3.2: Precision/Recall of different pattern matching models for function start identification.

Table 3.2 shows precision, recall, and runtime for each compiler and each function start identification model. From the table we can see that Rosenblum et al.'s implementation had an accuracy below 70%, while both ByteWeight-series models achieved an accuracy of more than 85%. Note that ByteWeight with 10-length and normalized signatures (the last row in table) performed particularly well, with an accuracy of approximately 97%, a more than 35% improvement over Rosenblum et al.'s implementation.

Table 3.2 also details the accuracy and performance differences among ByteWeight with different configurations. Comparing against the full configuration model (ByteWeight), the model with a smaller maximum signature length (ByteWeight (3)) performs slightly faster (3% improvement), but sacrifices 7% in accuracy. The model without signature normalization (ByteWeight (no-norm)) has only 1% higher precision but 6.68% lower recall, and the testing time is ten times longer than that of the normalized model.

### 3.4.3 Function Start Identification

The second experiment evaluated our full function start identification against existing static analysis tools. We compared ByteWeight (no-RFCR)—a version without recursive function call resolution, ByteWeight, and the following tools:

**IDA.** We used IDA 6.5, build 140116 along with the default FLIRT signatures. All function identification options were enabled.

**BAP-legacy.** We used BAP-legacy 0.7, which provides a `get_function` utility that can be invoked directly.

**Dyninst.** Dyninst offers the tool *unstrip* [114] to identify functions in binaries without debug information.

**Naive Method.** This matched simple `0x55` (`push %ebp` or `push %rbp`) and `0xc3` (`ret` or `retq`) signatures only.

We divided our data set into four categories: ELF x86, ELF x86-64, PE x86, and PE x86-64. Unlike the previous experiment, binaries from various compilers but the same target were grouped together. Overall, we had 1032 ELF x86 and ELF x86-64 binaries, and 68 PE x86 and PE x86-64 binaries. We evaluated these categories separately, and again applied 10-fold validation. During testing, we used the same score threshold $t = 0.5$ as in the first experiment.

Note that not every tool in our experiment supports all binary targets. For example, Dyninst does not support ELF x86-64, PE x86, or PE x86-64 binaries. We use “-” to indicate when the target is not supported by the tool. Also, we omitted 3 failures in ByteWeight, and 10 failures in Dyninst during this experiment. Due to a bug in BAP-legacy, ByteWeight failed in 3 `icc` compiled ELF x86-64 binaries: *ranlib* with `-O3`, *ld_new* with `-O2`, and *ld_new* with `-O3`. Dyninst

| | ELF x86 | ELF x86-64 | PE x86 | PE x86-64 |
|---|---|---|---|---|
| Naive | 0.4217/0.3089 | 0.2606/0.2506 | 0.6413/0.4999 | 0.0000/0.0000 |
| Dyninst | 0.8877/0.5159 | – | – | – |
| BAP | 0.8910/0.8003 | – | 0.3912/0.0795 | – |
| IDA | 0.7097/0.5834 | 0.7420/0.5550 | 0.9467/0.8780 | 0.9822/0.9334 |
| ByteWeight (no-RFCR) | 0.9836/0.9617 | 0.9911/0.9757 | 0.9675/0.9213 | 0.9774/0.9622 |
| ByteWeight | 0.9841/0.9794 | 0.9914/0.9847 | 0.9378/0.9537 | 0.9788/0.9798 |

Table 3.3: Precision/Recall for different function start identification tools.

failed in 8 `icc` compiled ELF x86-64 binaries and 2 `gcc` compiled ELF x86-64 binaries. The results of our experiment are shown in Table 3.3.

As evident in Table 3.3, ByteWeight achieved a higher precision and recall than ByteWeight without recursive function call resolution. ByteWeight performed above 96% in Linux, while all other tools all performed below 90%. In Windows, we have comparable performance to IDA in terms of precision, but improved results in terms of recall.

Interestingly, we found that the naive method was not able to identify any functions in PE x86-64. This is mainly because `VS` does not use `push %rbp` to begin a function; instead, it uses move instructions.

### 3.4.4 Function Boundary Identification

The third experiment evaluated our function boundary identification against existing static analysis tools. As in the last experiment, we compared ByteWeight, ByteWeight (no-RFCR), IDA, BAP-legacy, and Dyninst, classified binaries by their target, and applied 10-fold validation on each of the classes. The results of our experiment are shown in Table 3.4.

Our tool performed the best in Linux, and was comparable to IDA in Windows. In particular, for Linux binaries, ByteWeight and ByteWeight (no-RFCR) have both precision and recall above 90%, while IDA is below 73%. For Windows binaries, IDA achieves better results than ByteWeight with x86-64 binaries, but is slightly worse for x86 binaries.

|  | ELF x86 | ELF x86-64 | PE x86 | PE x86-64 |
|---|---|---|---|---|
| Naive | 0.4127/0.3013 | 0.2472/0.2429 | 0.5880/0.4701 | 0.0000/0.0000 |
| Dyninst | 0.8737/0.5071 | – | – | – |
| BAP | 0.6038/0.6300 | – | 0.1003/0.0219 | – |
| IDA | 0.7063/0.5653 | 0.7284/0.5346 | 0.9393/0.8710 | 0.9811/0.9324 |
| ByteWeight (no-RFCR) | 0.9285/0.9058 | 0.9317/0.9159 | 0.9503/0.9048 | 0.9287/0.9135 |
| ByteWeight | 0.9278/0.9229 | 0.9322/0.9252 | 0.9230/0.9391 | 0.9304/0.9313 |

Table 3.4: Precision/Recall for different function boundary identification tools.

### 3.4.5 Performance

**Training.** We compare ByteWeight against Rosenblum et al.'s work in terms of time required for training. Since we do not have access to either their training code or their training data, we instead compare the results based on the performance reported in paper. There are two main steps in Rosenblum et al.'s work. First, they conduct feature selection to determine the most informative idioms – patterns that either immediately precede a function start, or immediately follow a function start. Second, they train parameters of these idioms using a logistic regression model. While they did not provide the time for parameter learning, they did describe that feature selection required 150 compute *days* for 1,171 binaries. Our tool, however, spent only 586.44 compute *hours* to train on 2,064 binaries, including overhead required to setup cross-validation.

**Testing.** We list the performance of ByteWeight, IDA, BAP-legacy, and Dyninst for testing. As described in section 3.3, ByteWeight has three steps in testing: function start identification by pattern matching, function boundary identification by CFG and VSA, and recursive function call resolution (RFCR). We report our time performance separately, as shown in Table 3.5.

IDA is clearly the fastest tool for PE files. For ELF binaries, it takes a similar amount of time to use IDA and ByteWeight to identify function starts, however our measured times for IDA also include the time required to run other automatic analyses. BAP-legacy and Dyninst have better performance on ELF x86 binaries, mainly because they match fewer patterns than ByteWeight

|  | ELF x86 | ELF x86-64 | PE x86 | PE x86-64 |
|---|---|---|---|---|
| Dyninst | 2566.90 | – | – | – |
| BAP | 1928.40 | – | 3849.27 | – |
| IDA* | 5157.85 | 5705.13 | 318.27 | 371.59 |
| ByteWeight-Function Start | 3296.98 | 5718.84 | 10269.19 | 11904.06 |
| ByteWeight-Function Boundary | 367018.53 | 412223.55 | 54482.30 | 87661.01 |
| ByteWeight-RFCR | 457997.09 | 593169.73 | 84602.56 | 97627.44 |

* For IDA, performance represents the total time needed to complete disassembly and auto-analysis.

Table 3.5: Performance for different function identification tools (in seconds).

and do not normalize instructions. This table also shows that function boundary identification and recursive function call resolution are expensive to compute. This is mainly because we use VSA to resolve indirect calls during CFG recovery, which costs more than typical CFG recovery by recursive disassembly. Thus while ByteWeight with RFCR enabled has improved recall, it is also considerably slower.

## 3.5 Discussion

Recall that our tool considers a sequence of bytes or instructions to be a function start if the weight of the corresponding terminal node in the learned prefix tree is greater than 0.5. The choice to use 0.5 as the threshold was largely dictated by Rosenblum et al., who also used 0.5 as a threshold in their implementation. While this appears to be a good choice for achieving high precision and recall in our system, it is not necessarily the optimal value. In the future, we plan to experiment with different thresholds to better understand how this affects the accuracy of ByteWeight.

While there are similarities betwen Rosenblum et al.'s approach and ours, there are also several key differences that are worth highlighting:

- Rosenblum et al. considered sequences of bytes or instructions immediately preceding functions, called prefix idioms, as well the entry idioms that start a function. Our present model does not include prefix idioms. Rosenblumet al.'s experiments show prefix idioms increase accuracy in their model. In the future, we plan to investigate whether adding prefix matching to our model can increase its accuracy as well.

- Rosenblum et al.'s idioms are limited to at most $4$ instructions [1, p. 800] due to scalability issues with forward feature selection. With our prefix tree model, we can comfortably handle longer instruction sequences. At present, we settle on a length of 10. In the future, we plan to optimize the length to strike a balance between training speed and recognition accuracy.
- Rosenblum et al.'s CRF model considers both positive and negative features. For example, their algorithm is designed to avoid identifying two function starts where the second function begins within the first instruction of the first function (the so-called "overlapping disassembly"). Although we consider both positive and negative features as well, in constrast the above outcome is feasible with our algorithm.

While our technique is not compiler-specific, it is based on supervised learning. As such, obtaining representative training data is key to achieving good results with ByteWeight. Since compilers and optimizations do change over time, ByteWeight may need to be retrained in order to accurately identify functions in this new environment. Of course, the need for retraining is a common requirement for every system based on supervised learning. This is applicable to both ByteWeight and Rosenblum et al.'s work, and underscores the importance of having a computationally efficient training phase.

Despite our tool's success, there is still room for improvement. As shown in Section 4.5, over 80% of ByteWeight failures are due to the misclassification of the end instruction for a function, among which more than half are functions that do not return and functions that call such no-return functions. To mitigate this, we could backward propagate information about functions that do not return to the functions that call them. For example, if function `f` always calls function `g`, and `g` is identified as a no-return function, then `f` should also be considered a no-return function.

We could also use other abstract domains along with the strided intervals of VSA to increase the precision of our indirect jump analysis [22], which can in turn help us identify more functions more accurately.

One other scenario where ByteWeight currently struggles is with Windows binaries compiled with hot patching enabled. With such binaries, functions will start with an extra `mov %edi,%edi` instruction, which is effectively a 2-byte `nop`. A training set that includes binaries with hot patching can reduce the accuracy of ByteWeight. Because the extra instruction `mov %edi,%edi` is treated as the function start in binaries with hot patching, any subsequent instructions are treated as false matches. Thus, any sequence of instructions that would normally constitute a function start but now follows a `mov %edi,%edi` is considered to be a false match. Consider a hypothetical dataset where all functions start with `push %ebp; mov %esp,%ebp`, but half of the binaries are compiled with hot patching and thus start functions with an extra `mov %edi,%edi`. Half of the time, the sequence `push %ebp; mov %esp,%ebp` will be treated as a function start, but in the other half it will not be treated as such, thus leaving the sequence with a weight of 0.5 in our prefix tree. In order to deal with this compiler peculiarity, we would need give special consideration to `mov %edi,%edi`, treating both this instruction and the instruction following it as a function start for the sake of training.

Although training ByteWeight for function start identification is relatively fast, training for function *boundary* identification is still quite slow. Profiling reveals that most of the time is spent building CFGs, and in particular resolving indirect jumps using VSA. In future work, we plan to explore alternative approaches that avoid VSA altogether.

Finally, obfuscated or malicious binaries which intentionally obscure function start information are out of scope.

## 3.6 Related Work

In addition to the already discussed Rosenblum et al. [1], there are a variety of existing binary analysis platforms tackle the binary identification problem. BitBlaze [30] assumes debug information. If no debug information is present, it treats the entire section as one function. BitBlaze also provides an interface for incorporating Hex Rays function identification information.

Dyninst [62] also offers tools, such as *unstrip* [114], to identify functions in binaries without debug information. Within the Dyninst framework, potential functions in the `.text` section are identified using the hex pattern `0x55` representing `push %ebp`. First, Dyninst will start at the entry address and traverse inter- and intra-procedural control flow. The algorithm will scan the gaps between functions and check if `push %ebp` is present. This does not preform well across different optimizations and operating systems.

IDA using proprietary heuristics and FLIRT [55] technique attempts to help security researchers recover procedural abstractions. However, updating the signature database requires an amount of manual effort that does not scale. In addition, because FLIRT uses a pattern matching algorithm to search for signatures, small variations in libraries such as different compiler optimizations or the use of different compiler versions, prevent FLIRT from recognizing important functions in a disassembled program. The Binary Analysis Platform (BAP-legacy) also attempts to provide a reliable identification of functions using custom-written signatures [32].

Kruegel et al. perform exhaustive disassembly, then use unigram and bigram instruction models, along with patterns, to identify functions [72]. Jakstab uses two pre-defined patterns to identify functions for x86 code [71, §6.2].

## 3.7 Conclusion

We introduce ByteWeight, a system for automatically learning to identify functions in stripped binaries. In our evaluation, we show on a test suite of 2,200 binaries that ByteWeight outperforms previous work across two operating systems, two compilers, and four different optimizations. In particular, ByteWeight misses only 44,621 functions in comparison with the 266,672 functions missed by the industry-leading tool IDA. Furthermore, while IDA misidentifies 459,247 functions, ByteWeight misidentifies only 43,992 functions. To seed future improvements to the function identification problem, we are making our tools and dataset available in support of open science at `http://security.ece.cmu.edu/byteweight/`.

# Chapter 4

# Automatic Exploit Reuse for Control Flow Hijacking Exploits

This chapter of the dissertation focuses on automatic exploit reuse technique. At a high level, automatic exploit reuse is an active defense technique: a victim who receives an attack can have the ability to take advantage of the received attack and generate his own attack, especially for control flow hijacking exploits which is a critical type of exploit that could take the full control of the compromised machine. In this chapter, we describe how we initiate the automatic exploit reuse problem and introduce our design of the automatic exploit reuse technique for control flow hijacking exploits.

Remote exploits are extremely dangerous. With the help of remote exploits against a piece of software running on a victim computer, an attacker can install backdoors and exfiltrate sensitive information without physical access to the compromised system, leading to real-world impacts on the finances and reputation of the victim.

However, developing a remote exploit is not easy. A comprehensive understanding of the vulnerability is a must, and complex techniques to bypass defenses on the remote system are necessary. When possible, rather than developing a new exploit from scratch, attackers prefer to *reuse* existing exploits in their attacks, making necessary changes to adapt these exploits to new environments. One such adaptation is the replacement of the original *shellcode* (i.e., the attacker-injected code that is executed as the final step of the exploit) in the original exploit with a replacement shellcode, resulting in a modified exploit that carries out the actions desired by the attacker as opposed to the original exploit author. We call this a *shellcode transplant*. Shellcode transplanting has many applications, including reversing command and control protocols, understanding captured exploits, and replaying attacks. Thus, this capability is very helpful in situations ranging from rapid cyber-response (i.e., quick analysis of and response to 0-day attacks) and adversarial scenarios (like cyber-security Capture-The-Flag competitions or cyber warfare in the real world). Unfortunately, current techniques to transplant shellcode generally require an analyst to have a decent understanding of how the original exploit interacts with the program, what vulnerability it triggers, and how it bypasses deployed exploit mitigations. As a result, the analyst must put a lot of effort into development and debugging, which negates much of the advantage of shellcode transplanting.

In investigating this problem, we identified three main challenges to tackling the *shellcode transplant problem*. First, there is generally no clear boundary separating the shellcode from the rest of an exploit. Second, as an exploit's shellcode is commonly constructed through non-trivial data transformations, even if the bytes representing the original shellcode could be separated from the exploit, rewriting these bytes to a replacement shellcode would be non-trivial. Third, the shellcode and the remainder of the content in an exploit can be mutually dependent on each other (e.g., a field in the exploit payload may dictate the size of the embedded shellcode). Such relations can pose potentially complex constraints on any replacement shellcode that might be transplanted. When those constraints are violated by replacement shellcode, it is challenging to modify the exploit and/or the replacement shellcode in order for the modified exploit to function properly.

Previous work in the field of automated exploit generation generates exploits by constraining the memory bytes in each attacker-controlled buffer to the target shellcode. They enumerate all possible offsets in every attacker-controlled buffer until a solution is found [37, 63]. Such methods are insufficient. In the worst case, when attempting to compensate for the case of conflicting constraints on the replacement shellcode, these methods degenerate to a symbolic exploration of the program, which generally ends in a path explosion problem or is hampered by the inability of the symbolic execution engine to efficiently reverse complex data transformations. In fact, as we show in our evaluation, less than a third of the original exploits in our dataset can be modified by existing techniques.

In this chapter, we present *ShellSwap*, an automated system that addresses the *shellcode transplant problem*. ShellSwap takes an existing exploit and a user-specified replacement shellcode as input and produces a modified exploit that targets the same vulnerability as the original exploit does but executes the replacement shellcode after exploitation. ShellSwap tackles the challenges discussed above with a mix of symbolic execution and static analysis techniques, applying novel techniques to identify the original shellcode, recover the data transformation performed on it, and resolve any conflicts introduced by the transplant of the replacement shellcode. By utilizing information obtained from the original exploit and creatively transforming the replacement shellcode, ShellSwap rarely degrades to a pure symbolic exploration, and is thus more efficient and effective compared to previous solutions.Additionally, the use of carefully-designed systematic approaches enables ShellSwap to transplant more shellcode variants. In our experiment, ShellSwap successfully generates new exploits for 85% of all cases, which is almost three times the success rate of prior techniques.

To the best of our knowledge, ShellSwap is the first automated system that modifies exploits based on shellcode provided by analysts. In terms of offense, ShellSwap greatly reduces the overhead in attack reflection, which enables prompt responses to security incidents like 0-day attacks, especially in a time-constrained, competitive scenario such as a hacking competition or cyber warfare. ShellSwap also makes it possible for entities to *stockpile* exploits in bulk, and tailor them to specific mission parameters before they are deployed at a later time. As organizations such as the National Security Agency are commonly known to be stockpiling caches of vulnerabilities, such a capability can greatly reduce the overhead in using weapons from this cache. ShellSwap is also helpful in defense, where it can be used to debug exploits discovered in the wild (i.e. by transplanting a piece of shellcode that is benign or implements monitoring and reporting functionality) and rediscover vulnerabilities being exploited.

Specifically, our work makes the following contributions:

- We design the ShellSwap system, which is the first end-to-end system that can modify an observed exploit and replace the original shellcode in it with an arbitrary replacement shellcode. Our system shows that the automatic exploit reuse is possible: even a person who has little understandings about security vulnerabilities can retrofit an exploit for their custom use-case.
- We propose novel, systematic approaches to utilize information from the original exploit to prevent ShellSwap from degenerating to inefficient symbolic exploration, and revise the replacement shellcode without changing its semantics to fit constraints implicit to the original exploit. Those approaches are essential to the performance of ShellSwap.
- We evaluate our system on 100 cases — 20 original exploits, each with 5 different pieces of shellcode. Our system successfully generates modified exploits in 85% of our test set, and all new exploits work as expected. We also compare our system with the previous state of the art, and we find that previous methods only work for 31% of our test set. The fact that ShellSwap exhibits a success rate almost triple that of the previous solution implies that the impact of the challenges inherent in shellcode transplant were under-estimated, and that future work targeting this problem will be beneficial.

## 4.1 Overview

ShellSwap takes, as an input, a *vulnerable program*, the *original exploit* that had been observed being launched against this program, and a *replacement shellcode* that the *original shellcode* in the original exploit should be replaced with. Given these inputs, it uses a combination of symbolic execution and static analysis to produce a *modified exploit* that, when launched against the vulnerable program, causes the replacement shellcode to be executed.

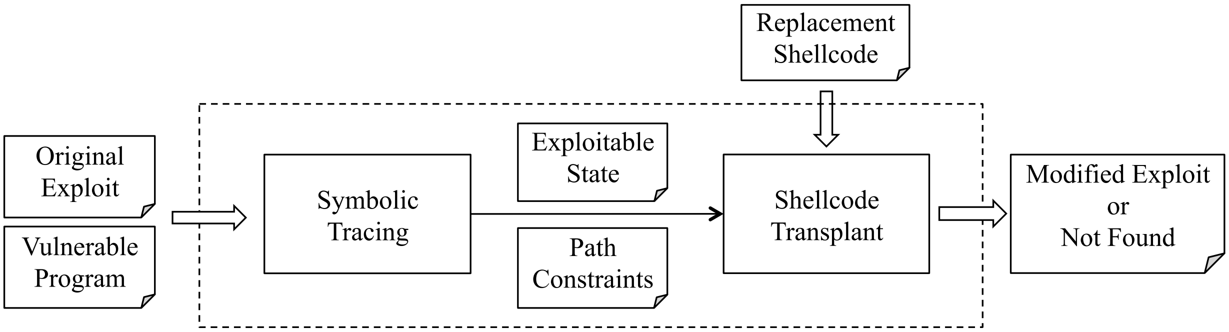


Figure 4.1: The architecture of the ShellSwap system.

Our intuition for solving the shellcode transplant problem comes from the observation that a successful control flow hijacking exploit consists of two phases: before the hijack, where the program state is carefully set up to enable the hijack, and after the hijack, when injected shellcode carries out attacker-specified actions. We call the program state after the first phase the *exploitable state*, and we call the instruction sequence that the program executes until the exploitable state the *exploit path*. An input that makes the program execute the same path as the original exploit does will lead the program to an exploitable state. Therefore, if we find an input that executes the instructions of the original exploit path in the first phase and the new shellcode in the second phase, that input represents the modified exploit.

Given these inputs, it proceeds through a number of steps, as diagrammed in Figure 4.1. The steps for generating the new exploit are as follows:

- **Symbolic Tracing.** The path generator replays the exploit in an isolated environment and records the executed instructions. The output of the path generator is a sequence of instruction addresses, which we call the *dynamic exploit path*.

  The path generator passes the dynamic exploit path to the symbolic tracing engine. Then tracer sets the input from the exploit as a symbolic value and starts symbolically executing the program. At every step of this execution, the tracer checks if the current program state violates a security policy. There are two reasons for this: a) we want to double check that the exploit succeeds, and b) we need to get the end of the normal execution and the start of malicious computation, where the exploit diverts the control flow of the program to the shellcode. When the tracer detects that the security policy has been violated, it considers the trace complete and the exploitable state reached.

  The tracing engine records the path constraints introduced by the program on the exploit input in order to reach the exploitable state, and the memory contents of the exploitable state itself. These will be used in the next step to surmount challenges associated with shellcode transplanting.

- **Shellcode Transplant.** Shellcode transplant is the critical step in the ShellSwap system. It takes the exploitable state, the path constraints, and the replacement shellcode as input, and outputs a modified exploit that takes advantage of the vulnerability and executes the replacement shellcode. After this step, the system will output either the modified exploit or an error indicating that a modified exploit could not be found.

These steps are further described in Section 4.2 (for Symbolic Tracing) and Section 4.3 (for Shellcode Transplant).

ShellSwap focuses on exploits against control-flow hijacking vulnerabilities, which are a type of software bug that allows an attacker to alter a program's control flow and execute arbitrary code (specifically, the shellcode). Control-flow hijacking vulnerabilities have been considered as the most serious vulnerabilities, since the attacker can take control of the vulnerable system. Unfortunately, control-flow hijacking vulnerabilities are the most prevalent class of vulnerabilities in the real world: over the past 18 years, 30.6% of reports in the Common Vulnerabilities and Exposures database represent control-flow hijacking vulnerabilities [42]. Thus, while the ability to reason about other types of exploits would be interesting, we leave the exploration of this to future work.

### 4.1.1 Motivating Example

To better communicate the concept of shellcode transplant and demonstrate the challenges inherent to it, we provide a motivating example. We first introduce a vulnerable program and an original exploit, and then discuss the challenges posed by two different instances of replacement shellcode.

**Vulnerable Program.** Consider a vulnerable program with source code shown in Code 4.1, where the program receives a string terminated by a newline, checks the first character and calculates the length of the string. Note that the source code is for clarity and simplicity; our system runs on binary program and does not require source code.

```
 1  int example(){
 2    int len = 0;
 3    char string[20];
 4    int i;
 5    if (receive_delim(0, string, 50, ’\n’) != 0)
 6      return -1;
 7    if(string[0] == ’^’)
 8        _terminate(0);
 9    for(i = 0; string[i] != ’\0’; i++)
10      len++;
11    return len;
12  }
```

Code 4.1: Motivating Example.

This program has a control-flow hijacking vulnerability in the processing of the received input. The `string` variable is a 20-byte buffer defined at line 3. However, the string received from user input can have up to 50 characters, which will overflow the buffer `string` and eventually overwrite the return address stored on the stack if the provided string is long enough. Figure 4.2 shows the stack layout of the `example` function. The saved return address (shown as saved `%eip`) is 36 bytes above the beginning of buffer `string`. This implies that if the received input has more than 36 characters, the input will overwrite the saved return address and change the control flow of the program when function `example` returns.

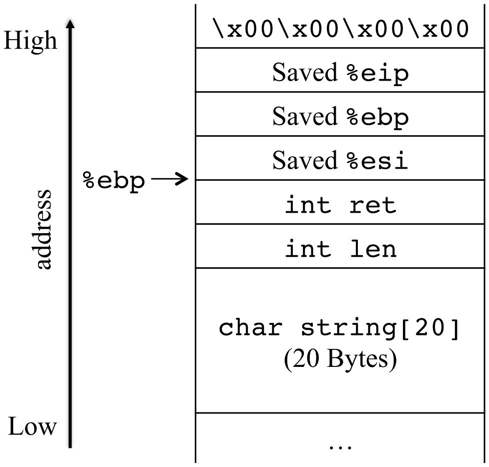


Figure 4.2: The stack layout of the `example` function.

```
shellcode = "\x31\xc0\x40\x40\x89\x45\xdc"
exploit = shellcode + "\x90" * (36 - len(shellcode)) + "\x50\xaf\xaa\
    xba\n"
```

Code 4.2: The original exploit with `shellcode`.

**Original Exploit.** Listing 4.2 shows the original exploit for the running example. The shellcode starts at the beginning of the exploit, followed by padding and the address with which to overwrite the return address. When the vulnerable program executes with the original exploit, the return address for function `example` will be changed to `0xbaaaaf50`, which points to the beginning of buffer `string`, and when function `example` returns, the control flow will be redirected to the shellcode.

### 4.1.2 Challenges

To demonstrate the challenges inherent in the shellcode transplant problem, we first consider a naive approach: if we find the location of the old shellcode in the original exploit, we could generate a new exploit by replacing, byte by byte, the old shellcode with the new one. We call this the *shellcode byte-replacement approach*. However, this naive approach assumes two things, that the shellcode stays in its original form throughout execution and that the replacement shellcode is the same size as the original shellcode. As we discussed previously, both of these assumptions are too strict for real-world use cases.

For example, consider the following replacement shellcode for the original exploit in our motivating example:

```
xor     %esi,%esi           ; 31 f6
lea     0x1(%esi),%ebx      ; 8d 5e 01
lea     0x8(%esi),%edx      ; 8d 56 08
push    0xaaaaaaaa          ; ff 35 aa aa aa aa
push    $0xdddddddd         ; 68 dd dd dd dd
mov     %esp,%ecx           ; 89 e1
lea     0x2(%esi),%eax      ; 8d 46 02
int     $0x80               ; cd 80
```

Code 4.3: The disassembly of the replacement shellcode `shellcode1`.

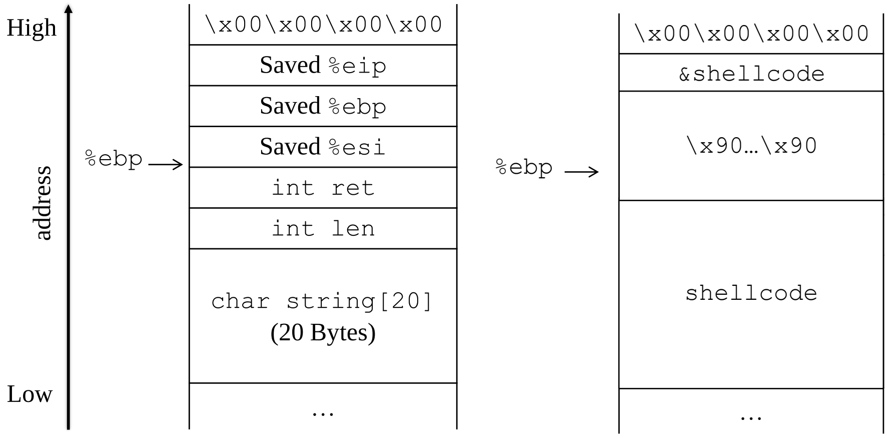
If we apply the shellcode byte-replacement method, the modified exploit be:

```
shellcode = "\x31\xf6\x8d\x5e\x01\x8d\x56\x08\xff\x35\xaa\xaa\xaa\xaa\
    x68\xdd\xdd\xdd\xdd\x89\xe1\x8d\x46\x02\xcd\x80"
exploit = shellcode + "\x90" * (36 - len(shellcode)) + "\x50\xaf\xaa\
    xba\n"
```

Code 4.4: The modified exploit for `shellcode1` using the shellcode replacement approach.

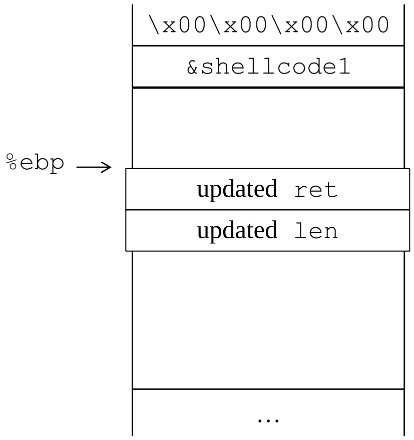
However, the modified exploit *will not work* when applied to our motivating example. Figure 4.3a shows the stack layout before function `example` starts. Besides saved registers, there are two variables between `string` and the saved `%eip`. When the program receives an input, the resulting stack layout is shown in Figure 4.3b. However, control is not immediately transferred to the shellcode. The program continues, and because variable `len` is updated before returning, the value at this address changes. By the time the function transfers control flow to the shellcode, the program changes the 20th through the 28th bytes of the replacement shellcode, as shown in Figure 4.3c. In our example, this represents unexpected modification to the replacement shellcode, rendering it nonfunctional.

In some cases, previous work is, using very resource-intensive techniques, capable of re-



(a) Before receiving an exploit.

(b) After receiving an exploit.



(c) When `example` returns.

Figure 4.3: The stack layout of function `example` at runtime.

finding the vulnerability and re-creating an exploit, but these systems all suffer from extreme scalability issues because they approach vulnerability detection as a *search problem*. If we do not want to re-execute these resource-expensive systems to re-identify and re-exploit vulnerabilities, a new approach is needed. To this end, we identified two main categories of challenges in shellcode transplanting: one dealing with the layout of memory at the time the vulnerability is triggered, and the other having to do with the actions taken in the path of execution *before* the vulnerability is triggered.

**Memory Conflicts.** Previous work [37, 63] places shellcode in memory by querying a constraint solver to solve the constraints generated in the Symbolic Tracing step and concretizing a region of memory to be equal to the desired shellcode. However, as is the case in our naive byte-replacement approach, this is not always possible: often, when dealing with fine-tuned exploits, there is simply not enough symbolic data in the state to concretize to shellcode [107].

For example, recall the shellcode in Listing 4.4 in the context of our motivating example. This piece of shellcode is 26 bytes long, which *should* have fit into the 50 bytes of user input. However, the 20th through the 28th byte are overwritten, and the 36th through 40th byte must be set to the address of the shellcode (to redirect control flow). This leaves three symbolic regions: a 20-byte one at the beginning of the buffer, an 8-byte one between the `ret` and `len` variables and the saved return address, and the 10 bytes after the saved return address. None of these regions are big enough to place this shellcode, causing a *memory conflict* for the shellcode transplanting process.

**Path Conflicts.** To drive program execution to the exploited state, the content of the modified exploit must satisfy the path constraints recovered from the Symbolic Tracing step. However, by requiring the replacement shellcode to be in the memory of the exploitation state, we add new constraints ("shellcode constraints") on the exploit input. These new conditions may be conflict with those generated along the path. We call such conflict the *path conflict*. In the presence of such a conflict, if we locate the replacement shellcode in the exploitation state (and discard the path constraints that conflict with this), the exploit path will change, and the new program state resulting from the changed path may not trigger the vulnerability.

For example, consider the replacement shellcode in Listing 4.5 in the context of the motivating example.

```
push    $0x0                    ; 6a 00
push    $0xa65                  ; 68 65 0a 00 00
push    $0x646f636c             ; 68 6c 63 6f 64
push    $0x6c656873             ; 68 73 68 65 6c
mov     $0x2,%eax               ; b8 02 00 00 00
mov     $0x1,%ebx               ; bb 01 00 00 00
mov     %esp,%ecx              ; 89 e1
mov     $0xa,%edx              ; ba 0a 00 00 00
lea     0x10(%esp),%esi        ; 8d 74 24 10
int     $0x80                  ; cd 80
```

Code 4.5: The disassembly of the replacement shellcode `shellcode2`.

When the running example executes with an input string, the `for` loop body before the return increments `i` until `string[i]` is a null byte. For the original exploit, the loop will repeat for 40 times (the length of the exploit string), meaning that the path constraints will mandate that the first 40 bytes of `string` are not null. For the replacement shellcode, however, if we locate the new shellcode at the beginning of `string`, the loop will only iterate once, because the second byte of the shellcode is null. This creates a contradiction between the path constraints and the shellcode constraints.

**Surmounting the Challenges.** The intelligent reader can certainly envision approaches to achieve shellcode transplanting in the motivating example. However, this example is just 12 lines of code. One can see that, with bigger examples and in the general case, these challenges can be quite complicated to surmount.

In the rest of the chapter, we will discuss how to identify conflicts while transplanting the shellcode and how to satisfy both memory and path conflicts to successfully transplant shellcode in a wide variety of exploits.

## 4.2 Symbolic Tracing

Essentially, ShellSwap separates the entire execution of the original exploit into two phases: before the control-flow hijack and after the control-flow hijack. The Symbolic Tracing step analyzes the former. The goal of this step is to generate the *exploitable state* of the program and record the *path constraints* that are induced by conditional branches that are encountered on the path. This involves two main considerations.

First, we must determine when the control-flow hijack occurs. We do this by leveraging the concept of *security policies*, which has been thoroughly explored by researchers [33, 50, 67, 75, 117]. In our work, we use the well-studied taint-based enforceable security policy [96, 117]. This policy determines whether or not a program state is safe by checking the instruction being executed. If the instruction directly is tainted by remote input, then the program state is deemed unsafe and the path is terminated.

Second, we must determine how to perform the tracing, as there are several possible techniques that might be used here. For example, we could use dynamic taint analysis to identify when executed instructions are tainted by input data. While this would be relatively fast, taint analysis is not sufficient. Although it can identify violations to our security policy caused by tainted input, it cannot recover and track path constraints. Thus, in our system, we apply concolic execution to trace the path that the exploit runs on the program. We ensure tracing accuracy in two ways: we record a dynamic trace of the exploit process (and require that our symbolic trace conform to the same instructions), and we *pre-constrain* the symbolic data to be equal to the original exploit. The former avoids the path explosion inherent in concolic execution exploration (because we only care about the branch that the exploit chooses), and the latter greatly simplifies the job of the symbolic constraint solver during tracing (by providing it with a pre-determined solution). This method is similar to the pre-constraint tracing and the input pre-constraining approach proposed by Driller [110] (and, in fact, part of the implementation derives off of Driller's tracing module).

The trace-directed symbolic execution takes a program and an original exploit and produces path constraints and the exploitable state. The exploitable state includes the symbolic value of registers and memory at the moment that the program starts to execute the shellcode. After this step completes, the pre-constraints introduced in the beginning are removed, making it possible to constrain some of the memory in the exploitable state to contain values representing, for example, the replacement shellcode. The remaining path constraints guarantee that any satisfying input will make the program to execute the same execution trace and triggers the vulnerability.

## 4.3 Shellcode Transplant

After the exploitable state and the path constraints associated with it have been recovered, ShellSwap can attempt to *re-constrain* the shellcode to be equal to the replacement shellcode by adding *shellcode constraints*. However, as discussed in Section 4.1, the shellcode constraints may conflict with the path constraints. Previous work [37, 63] addresses this issue by trying other shellcode locations, but even the simple motivating example in Section 4.1 is too complicated for this to work.

The Shellcode Transplant steps attempts to resolve these conflicts. If it can do so, the modified exploit, containing the replacement shellcode, is produced. If it fails, it returns an error indicating that the exploit could not be found.

The step proceeds in several phases, in a loop, as shown in Figure 4.4. First, in the *Preprocessing* phase, ShellSwap identifies possible memory locations into which replacement shellcode (or pieces of it) can be placed. Next, in the *Layout Remediation* phase, it attempts to remedy memory conflicts (as discussed in Section 4.1.2) and fit the replacement shellcode into the identified memory locations, performing semantics-preserving modifications (such as code splitting) if necessary. If this fails due to a resulting conflict with the path constraints (a path conflict, as discussed in Section 4.1.2), ShellSwap enters the *Path Kneading* phase and attempts to identify alternate paths that resolve these conflicts while still triggering the vulnerability. If such a path can be found, its constraints replace the path constraints, and the system repeats from the preprocessing phase.

If ShellSwap encounters a situation where neither the memory conflicts nor the path conflicts can be remedied, it triggers the *Two-Stage Fallback* and attempts to repeat the Shellcode Transplant stage with a fallback, two-stage shellcode.

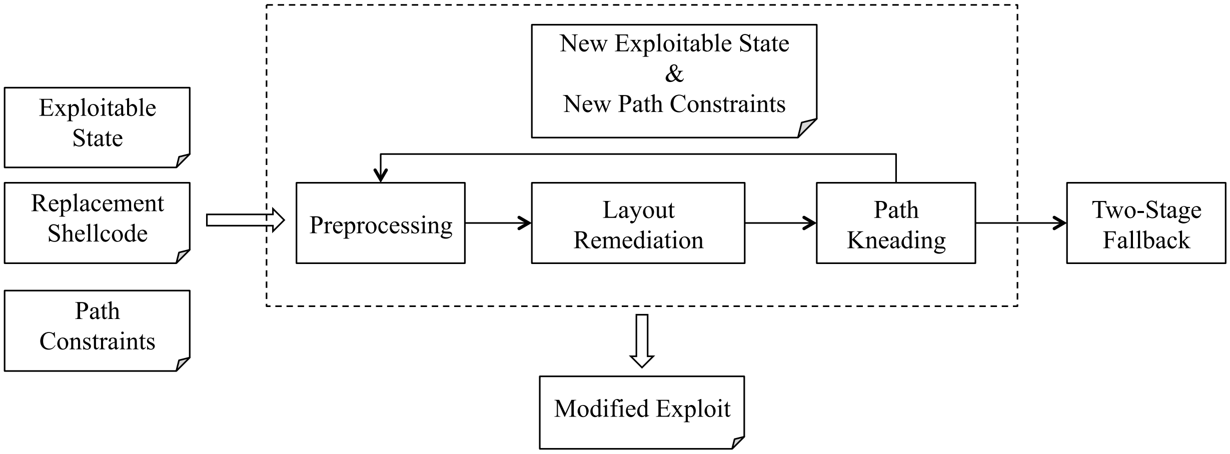


Figure 4.4: The phases of the Shellcode Transplant step.

### 4.3.1 Preprocessing

Before the system tries to locate the new shellcode, it scans the memory in the exploitable state to identify *symbolic buffers*. A symbolic buffer is a contiguous memory where all bytes are symbolic. To find symbolic buffers, our system iterates the bytes of the memory, marking each contiguous region. After finding all symbolic buffers, we sort the buffers by the length and the number of symbolic input variables involved in each buffer. Buffers with bigger length and more symbolic values has more varieties of concrete values, and thus are more likely to be able to hold the replacement shellcode.

### 4.3.2 Layout Remediation

Given symbolic buffers from the previous phase, the system attempts to fit the replacement shellcode into the exploitable program state. As an innovation over prior work, ShellSwap does not consider a piece of shellcode as an integrated memory chunk. Instead, we model the new shellcode as a sequence of instructions. It is not necessary to keep these instructions contiguous; we could insert `jmp` instructions to “hop” from one shellcode instruction in one symbolic buffer to another instruction in another buffer. Thus, we attempt to fit pieces of the shellcode (plus any necessary jump instructions) into previously-identified symbolic buffers.

```
Input :
  $SH$: The new shellcode
  $ST$: The current exploitable program state. $ST.mem[j]$ means the memory at $j$ in the state $ST$
  $I$: The symbolic buffers generated by preprocessing
  $C$: The constraints set
  $i$: The index of the instruction of the shellcode
  $a$: The start address that we plan to put $SH[i]$
Output:
  $E$: A new exploit or Not Found
if $i > len(SH)$ then
    // We have successfully put the entire piece of shellcode to the exploitable state.
    $E \leftarrow$ Solve($C$);
    return $E$;
end
else if $i < 0$ then
    // We cannot successfully put the entire piece of shellcode to the exploitable state if we put $SH[i]$ at $a$.
    return Not Found;
end
else
    if $I$ has enough space after $a$ then
        // Construct the new constraint asserting the memory at $a$ concretize to the i-th byte of the replacement shellcode.
        $c \leftarrow (ST.mem[a : a + len(SH[i])] == SH[i])$;
        $C' \leftarrow C + c$;
        if Solve(C') has solution then
            $ST' \leftarrow$ a new state with $ST.mem[a : a + len(SH[i])] = SH[i]$;
            $a' \leftarrow$ Next($I, a + len(SH[i])$);
            return Locate($SH, ST', I, C', i + 1, a'$);
        end
        else
            // We cannot put $SH[i]$ at $a$. Instead, we need to find another location for $SH[i]$ and hop to the location.
            if $Hop(SH, ST, I, C, i, a) ==$ Not Found then
                return Not Found;
            end
            else
                $ST', a', C' \leftarrow$ Hop($SH, ST, I, C, i, a$);
                return Locate($SH, ST', I, C', i + 1, a'$);
            end
        end
    end
    return Not Found;
end
```

**Algorithm 1:** The algorithm of the `Locate` function.

**Input :**
$SH$: The new shellcode
$ST$: The current exploitable program state. $ST.mem[j]$ means the memory at $j$ in the state $ST$
$I$: The symbolic buffers generated by preprocessing
$C$: The constraints set
$i$: The index of the instruction of the shellcode. $SH[i]$ means the bytes for the i-th instruction of the shellcode $SH$.
$a$: The start address that we plan to put $SH[i]$
**Output:**
$ST'$: The updated exploitable program state, with the jump instruction and SH[i] in the memory.
$C'$: The updated constraints set
$a'$: The start address for the next instruction

```
if $i < 0$ then
    // We cannot successfully hop $SH[i]$.
    return Not Found;
end
else
    // find an address to put $SH[i]$
    $a' \leftarrow$ None;
    $a_t \leftarrow$ Next($I, a + len_{jmp}$));
    while $a_t$ is not None do
        $c \leftarrow (ST.mem[a_t : a_t + len(SH[i])] == SH[i])$;
        $C' \leftarrow C + c$;
        if Solve(C') has solution then
            // $SH[i]$ can be put at $ST.mem[a_t]$
            $c_{jmp} \leftarrow$ jump instruction constraint;
            $C'' \leftarrow C' + c_{jmp}$;
            if Solve($C''$) has solution then
                // The jump instruction can be put at $ST.mem[a]$
                $ST' \leftarrow$ a new state with $SH[i]$ and jump instruction;
                $a' \leftarrow$ Next($I, a_t + len(SH[i])$));
                return $ST', a', C'$;
            end
        end
        else
            $a_t \leftarrow$ Next($I, a_t$));
        end
    end
    // We cannot hop to an address with $SH[i]$ after address $a$.  Then
       we roll back and hop to the previous instruction.
    $ST', a', C' \leftarrow$ Rollback(SH, ST, C, I, a);
    return Hop($SH, ST', I, C', i - 1, a'$);
end
```

**Algorithm 2:** The algorithm for the `Hop` function.

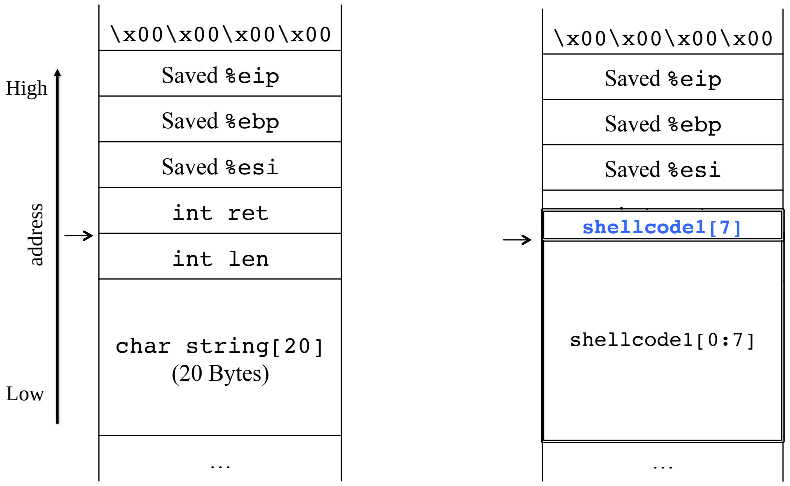
Algorithm 1 and Algorithm 2 shows the algorithms for Layout Remediation. The system invokes function `Locate`, and function `Locate` calls out to function `Hop` when needed. Both functions take five arguments as input: $SH, ST, I, C, i, a$, where $SH$ is the shellcode, $ST$ is the exploitable state, $I$ is the symbolic buffers, $C$ is the set of constraints for $ST$, $i$ is an index into the not-yet-written bytes of the replacement shellcode, and $a$ is the memory address being currently considered by the algorithm.

We use the motivating example to demonstrate how the algorithm works. As mentioned in Section 4.1.2, there will be three symbolic buffers in this example. Suppose the ShellSwap system tries to fit the shellcode from Listing 4.3 to the stack of the exploitable state of the motivating example. It calls `Locate` with $i = 0$ and $a =$ `&string`, initially trying to put the first instruction of the replacement shellcode at the beginning of the buffer `string`.

The layout remediation process is shown in Figure 4.5. As the first 6 instructions of the replacement shellcode satisfy the constraints in memory, the process will continue adding new instructions until the 7th instruction (Figure 4.5b). At this point, the system fails to add the 7th instruction (because `len` is in the way), so it calls function `Hop`, trying to jump over `len` and place the 7th instruction to into the next symbolic buffer (Figure 4.5c). In function `Hop`, it successfully finds a location for the 7th instruction. However, the `jmp` instruction cannot fit after the first 6 instructions (Figure 4.5d, so we roll back and call `Hop` to re-locate the 6th instruction (Figure 4.5e). Since the `jmp` instruction still covers `len`, this rollback occurs again, until the *5th* instruction ends up relocated, and a `jmp` inserted after the 4th instruction to the 5th instruction. In the end, this is repeated until the full shellcode is placed in memory, split into three parts as shown in Figure 4.5f.

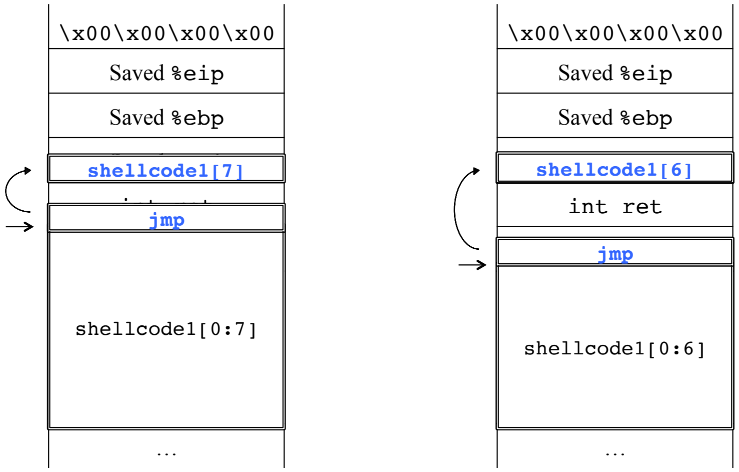
### 4.3.3 Path Kneading

If the system cannot find a new exploit for the new shellcode using the exploitable state of the original exploit, we need to diagnose the cause of conflict and tweak the path to generate new



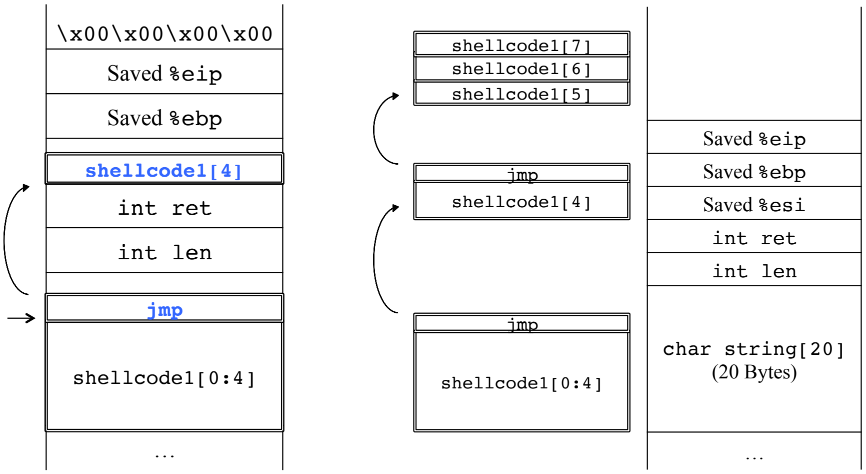
(a) The initial stack.

(b) `Locate`, $i = 7$, $a = \&\texttt{string} + 24$



(c) `Hop`, $i = 7$, $a = \&\texttt{string} + 24$

(d) `Hop`, $i = 6$, $a = \&\texttt{string} + 21$



(e) `Hop`, $i = 4$, $a = \&\texttt{string} + 14$

(f) The final shellcode layout.

Figure 4.5: The layout remediation process for the motivating example with `shellcode1`.

exploitable states and new path constraints. To diagnose the cause of conflict, we first identify the conflicting path constraints and then check which instructions generated them.

Since shellcode is placed to the exploitable state instruction by instruction, we can retrieve the smallest set of shellcode constraints that cause a path conflict as soon as `Locate` terminates unsuccessfully. Let $c$ be the constraint for locating the current instruction, and let $C$ be the set of path constraints set of the current state. We already know that $c$ and $C$ are conflicted (otherwise, a location for the last instruction would have been found), which implies that $c \wedge C = \mathit{False}$. To understand the cause of the conflict, we find the smallest set of path constraints $S$ such that: $S \subseteq C$, such that:

$$c \wedge S = \mathit{False} \text{ and } c \wedge (S - C) = \mathit{True}.$$

After finding the conflict subset, ShellSwap identifies the source of each constraint in this subset by checking the execution history for when it was introduced. If the conflicting constraint was introduced by condition branch, ShellSwap will tweak the path to *avoid* the path constraints in the conflict subset. The intuition for this is as follows: if the shellcode constraint contradicts a path constraint, then the shellcode constraint does *not* contradict the *negation* of that path constraint. For path constraints created by conditional branches, our idea is to negate the conflict path constraints by selecting the other branch in the program. In this way, if the program executes along the path with the opposite branch, the new path constraints will contain the negation of the previously-conflicting path constraint, and the new path constraints will not conflict with the shellcode constraint $c$.

For example, consider the motivating example and the replacement shellcode in Listing 4.5. As we described in Section 4.1.2, we encounter a path conflict because the `for` loop in our example, which runs 40 times for the original shellcode, only runs once for the replacement shellcode. Let $E$ be the exploit, and let $E_i$ be the $i$-th byte in $E$. The symbolic value of `string` in the exploitable state is equal to:

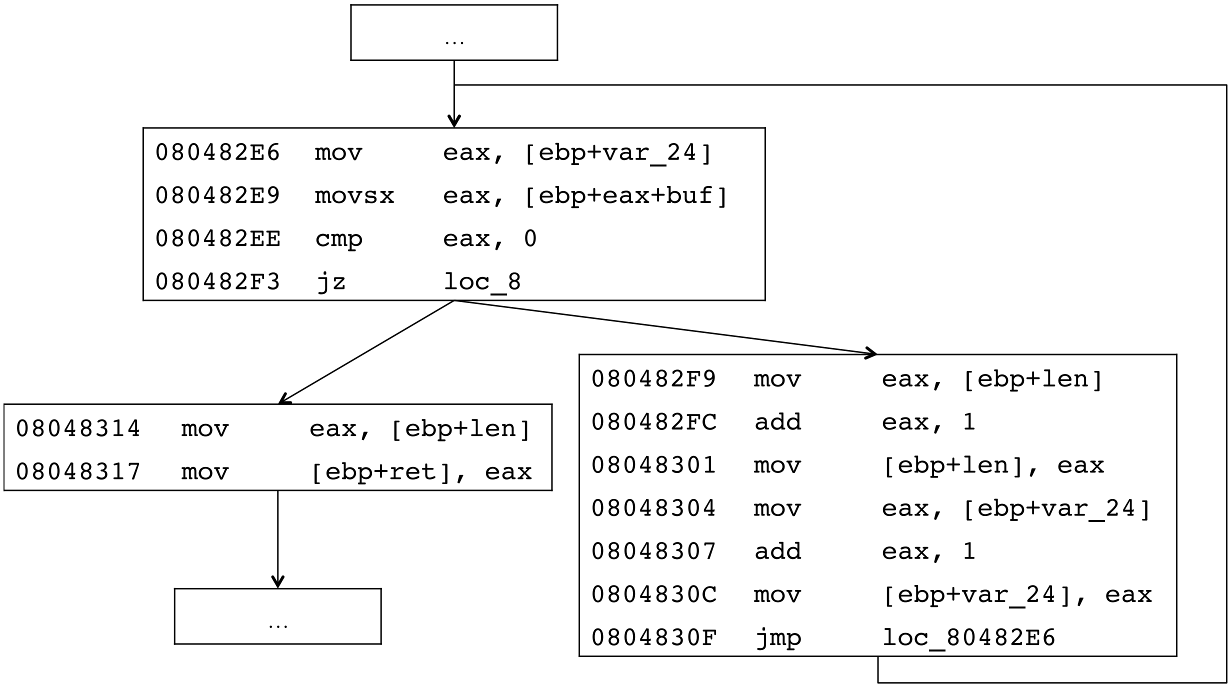
$$\texttt{Concatenate(}E_0\texttt{, }E_1\texttt{, ... , }E_{18}\texttt{, }E_{40}\texttt{)}$$

which means the string from the 0th to the 40th byte of the input. In this case, the path constraints include the following:

$$E_0 \neq \mathtt{'\backslash x00'} \land E_1 \neq \mathtt{'\backslash x00'} \land \ldots \land E_{40} \neq \mathtt{'\backslash x00'}$$

However, because the second character of the replacement shellcode is `'\x00'`, the shellcode constraints conflict with the path constraints[1].

Suppose that ShellSwap identifies this situation while trying to place the first instruction of the replacement shellcode at the beginning of `string`. After analyzing the conflicting constraint subset, we know that the conflict stems from the path constraint $E_1 \neq \mathtt{'\backslash x00'}$, and this constraint is created at address `0x080482F3`, shown in Figure 4.6. Specifically, the conflict constraint occurs at the second iteration of the for loop.



```
...
```

```
080482E6   mov     eax, [ebp+var_24]
080482E9   movsx   eax, [ebp+eax+buf]
080482EE   cmp     eax, 0
080482F3   jz      loc_8
```

```
08048314   mov     eax, [ebp+len]
08048317   mov     [ebp+ret], eax
```

```
...
```

```
080482F9   mov     eax, [ebp+len]
080482FC   add     eax, 1
08048301   mov     [ebp+len], eax
08048304   mov     eax, [ebp+var_24]
08048307   add     eax, 1
0804830C   mov     [ebp+var_24], eax
0804830F   jmp     loc_80482E6
```

Figure 4.6: Part of the control flow graph for the motivating example.

To generate a new path, we negate the conditional jump associated with the conflicting path constraint by modifying the trace to force an exit from the loop after the second iteration. How-

---

[1] The inquisitive reader might question why the part of the shellcode with the null byte could not be written *after* the return address to bypass this loop. However, a closer look at the replacement shellcode would reveal null bytes in many other locations as well.

ever, after this change, we need to merge the diversion back to the original path. We accomplish this by leveraging static analysis. First, we find the function containing the divergence point, and build a control flow graph for the specific function. Next, we statically find the descendants of the diverted node and see if any of the descendants appear in the original path after the negated node. For each satisfying descendant, we attempt to construct a new path that is identical to the original path until the negated node, followed by the detected detour back to the descendent node that appears in the original path, and then ending with the postfix from the descendant node to the end of the original path.

Figure 4.7 shows the generation of a new path. Suppose node $n_c$ is negated to $n_{c'}$, and node $n_d$ is the descendant of node $n_{c'}$. For the new path, the basic blocks do not change before $n_{c'}$ or after $n_d$. In between, we insert an intraprocedural path from $n_{c'}$ to $n_d$ G($n_{c'}$, $n_d$), which can be generated using the control graph of the function. In the best case, the question is equivalent to finding a path between two nodes in a directed graph. However, it is possible that there is no such path to rejoin the original path, or that the problem reduces to symbolic exploration (if the divergence is too big). In this case, ShellSwap falls back on the Two-Stage Fallback.

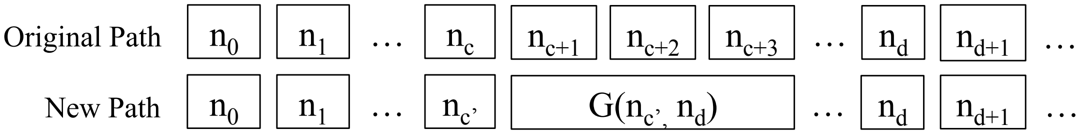


Figure 4.7: The generation of a new path. G(x, y) means a path between node x and y found by static analysis.

In the motivating example, as simply exiting the loop already rejoins the original path, the detour back to the path is trivial: it is the direct jump to the return site of the `example` function.

After constructing the new path, the ShellSwap system generates the new exploitable program state and a new set of path constraints using the Symbolic Tracing step. Meanwhile, it also checks if the new program state is still exploitable. If the new program state is exploitable, our system starts again from the preprocessing phase to fit the replacement shellcode into the new exploitable program state. Otherwise, the system will attempt to construct the other paths and generate the other program states, falling back on the Two-Stage Fallback if it is unable to do so.

### 4.3.4 Two-Stage Fallback

If ShellSwap is unable to overcome the memory and path conflicts and fit the replacement shellcode into the exploitable state, then it falls back on pre-defined a two-stage shellcode instead of the provided replacement shellcode. The motivation of this fallback is straightforward: if the provided shellcode cannot fit the exploitable state, even after Path Kneading, we try a smaller first-stage replacement shellcode that can then load an arbitrary second-stage shellcode.

There are several options for a first-stage shellcode. One option is a shellcode that reads shell commands from the socket and executes them. Another, to bypass modern defenses such as Data Execution Protection, could read a Return Oriented Programing payload over the stack and initiates a return. For our prototype, we implemented a stack-based shellcode-loading first-stage payload that reads a second-stage payload onto the stack and jumps into it. While this is not immune from DEP techniques, it is only meant as a proof of concept for our prototype.

Consider the motivating example. The program receives input by using the DECREE syscall `receive()` (more information on DECREE is provided in Section 4.5), which is a system call similar to `recv()` in Unix/Linux. If the new shellcode is longer than 50 bytes, we cannot generate a new exploit because the program is able to receive 50 bytes at most. In this case, we could consider the following template for generating a two-stage shellcode:

```
xor    %eax,%eax        ; 31 c0
inc    %eax             ; 40
inc    %eax             ; 40
inc    %eax             ; 40
xor    %ebx,%ebx        ; 31 db
inc    %ebx             ; 43
mov    %esp,%ecx        ; 89 e1     ecx: &dst
mov    _ ,%edx          ; 8b _      edx: len
mov    _,%esi           ; 8b _      esi: &ret
int    $0x80            ; cd 80
jmp    *%esp            ; ff e4
```

Code 4.6: The disassembly of the template for a two-stage shellcode.

This first-stage shellcode reads a string, stores at the bottom of the stack (`%esp`) and jumps to the received string. There are two blanks in the template – we need to fill the receiving length and the address of return value for register `%edx` and `%esi`, respectively. After completing the template, our system will restart the layout remediation process with the two-stage shellcode as the replacement shellcode. If the system cannot find a modified exploit using the Two-Stage Fallback, it returns an error indicating that no modified exploit could be found.

Although the two-stage shellcode helps to solve the shellcode transplant problem by increasing the situations in which ShellSwap can function, we consider this purely as a fallback. This is because two-stage exploits may be less robust than the other exploits, as they assume that the victim machine can receive extra bytes from the attacker. This assumption does not always hold. For instance, the victim machine may be protected by other mechanisms which block the message, such as an external firewall, or the network connection over which communication happens might already be closed when the vulnerability triggers. Therefore, our system prioritizes the conflict resolution approaches, and it will not trigger the Two-Stage Fallback when the previous layout remediation process fails.

## 4.4 Implementation

ShellSwap is implemented on top of `angr` [107], a binary analysis platform. We rely on angr's symbolic tracing component [7], which also leverages the QEMU emulator [5] for exploit replay and symbolic tracing. The core of our system, consists of about 2000 lines of Python code.

### 4.4.1 Finding Infeasible Constraint Sets

Finding a minimal subset of infeasible constraints, which is an essential part of Path Kneading, is not a trivial problem. The underlying constraint solver Z3, which is used in angr (and thus in ShellSwap), provides an `unsat_core` function to retrieve the *smallest* subset of an unsatisfiable set of constraints. However, in our experiment, we found that `unsat_core` can be very time consuming, and sometimes even lead to crashes of Z3. Since we weren't able to pinpoint the root cause of the problem, we further implement a constraint set slimming method (as described below) to resort to in case `unsat_core` fails.

The constraint set slimming is a divide-and-conquer approach. Given a constraint set $A$ and a constraint $c$ that contradicts $A$, constraint set slimming will try to find a subset of constraints in $A$ (but not the smallest subset) that still contradicts $c$. We first divide $A$ into two subsets and check if any of them is contradictory to constraint $c$. If both subsets contradict $c$, the final infeasible constraint set will include conflicting constraints subsets from the two. If only one subset contradicts $c$, the other subset can be safely discarded as the result will only contain conflicting constraints from the contradictory subset. We repeat this procedure on contradictory subsets recursively until we find the very last contradictory subset, which either contains a single constraint that contradicts $c$, or several constraints that none of which contradicts $c$ if considered individually. The union of all conflicting subsets of constraints represent the slimmed set of constraints.

### 4.4.2 Optimizations

Much of the execution in symbolic tracing does not involve symbolic data. To speed up the tracing step, ShellSwap enables code JIT'ing (through the use of Unicorn Engine [90]) by default, which allows instructions in the original exploit to be executed natively instead of being emulated. While it greatly speed up symbolic tracing, we find that this step is still the bottleneck in ShellSwap: as discussed in Section 4.5, an average of 95% of execution time is spent in this step.

To avoid generating an entire control-flow graph in our path kneading component, we used a fast function detection approach to pick out the exact function for which to generate the control flow [23].

In the course of the development of this system, we have upstreamed many big-fixes and some improvements to angr and its tracing module. With these fixes, we observed a *1000-times* speed improvement on some samples in our evaluation.

## 4.5 Evaluation

In this section, we present our evaluation of ShellSwap. We first describe the data set, including all vulnerable programs and exploits, used in our evaluation (Section 4.5.1). Then, we show the experimental setup in Section 4.5.2. Next, we demonstrate the effectiveness of our approach in Section 4.5.3 by evaluating both ShellSwap and a reference implementation of previous work on 20 original exploits and 5 pieces of replacement shellcode. There, we show the necessity of ShellSwap in effectively transplanting shellcode. In the end, we evaluate the efficiency of ShellSwap and display the results in Section 4.5.4.

### 4.5.1 Data Set

Our evaluation data set contains three parts: 11 vulnerable binaries, 20 original exploits, and 5 pieces of replacement shellcode. We present how the data set is constructed below.

**Vulnerable Binaries.** We selected 11 vulnerable binaries (see Table 4.1) from the qualifying event as well as the final event of DARPA Cyber Grand Challenge (CGC). These binaries are shipped with source code, reference exploits, and actual exploits generated by other CGC participants, making them a perfect fit for our evaluation. All of the binaries are standalone x86 binaries with a special set of system calls (DECREE syscalls), roughly analogous to the Linux system calls `recv` (as DECREE's `receive`), `send` (as DECREE's `transmit`), `mmap` (as DECREE's `allocate`), `munmap` (as DECREE's `deallocate`), `select` (as DECREE's `fdwait`), `get_random` (as DECREE's `random`), and `exit` (as DECREE's `_terminate`). Sizes of those binaries range from 83 KB to 18 MB. Those vulnerable binaries cover a wide range of subtypes of control flow hijack vulnerabilities, including stack overflow, heap overflow, integer overflow, arbitrary memory access, improper bound checking, etc.

**Exploits.** As the CGC provides generators for reference exploits, we generated a few exploits for each vulnerable binary, for a total of 20 reference exploits (as is shown in Table 4.1). It is worth noting that exploits (or *Proofs of Vulnerability* in CGC terminology) in CGC are special in the sense that each of them should demonstrate attacker's ability to fully control values in two registers: the instruction pointer and one other register. As a result, some generated exploits do not contain any shellcode. We manually post-processed all exploits to make sure each one of them has a piece of shellcode to execute in the end of the exploitation.

**Shellcode.** As shown in Table 4.2, we collected five instances of replacement shellcode from three different sources, four of which are from CGC finalists (ForAllSecure and Shellphish), and one of which is manually crafted by ourselves. This range of replacement shellcode instances is important: with the shellcode coming from multiple sources, we can mimic the setting of cyber attack customization in our experiments. We refer to these instances as $S_1$ through $S_5$. Therefore, with five instances of replacement shellcode for each of the 20 original exploits in our dataset, we have a total of 100 modified exploits for ShellSwap to generate.

### 4.5.2 Experiment Setup

One of the applications of transplanting shellcode is to automatically reflect, or *ricochet*, an attack coming from a rival. In this scenario, the victim first detects an exploit coming from the attacker. They then automatically replace the payload (the shellcode) in the exploit and replay the modified exploit against the attacker. We try to simulate such a scenario in our experiment, where the attacker emits original exploits and the victim (or replayer/reflector) replays a modified exploit with the shellcode replaced.

| **Binary** | **Size** | **Vulnerability Type** | **Original Exploits** | **Modified Exploits** | **SystemM** | **ShellSwap** |
|---|---|---|---|---|---|---|
| CADET_00001 | 83 KB | Buffer Overflow | 1 | 5 | 4 | 5 |
| CROMU_00001 | 92 KB | Integer Overflow | 1 | 5 | 3 | 5 |
| EternalPass | 18 MB | Untrusted Pointer | 2 | 10 | 0 | 8 |
| Hug_Game | 3.1 MB | Improper Bounds Checking | 1 | 5 | 0 | 2 |
| LUNGE_00002 | 1.6 MB | Off-by-one Error | 1 | 5 | 0 | 5 |
| On_Sale | 125 KB | Buffer Overflow | 4 | 20 | 14 | 20 |
| OTPSim | 106 KB | Improper Bounds Checking | 2 | 10 | 10 | 10 |
| Overflow_Parking | 92 KB | Integer Overflow | 1 | 5 | 0 | 0 |
| SQL_Slammer | 102 KB | Buffer Overflow | 3 | 15 | 0 | 12 |
| Trust_Platform_Module | 89 KB | Buffer Overflow | 3 | 15 | 0 | 15 |
| WhackJack | 105 KB | Buffer Overflow | 1 | 5 | 0 | 3 |
| **Total** | | | 20 | 100 | 31 | 85 |

Table 4.1: This table shows the vulnerable binaries, the types of their vulnerabilities, the numbers of original exploits of each binary, the total number of attempted exploit modifications (one replacement shellcode per original exploit per binary), and the number of modified exploits successfully produced by SystemM and ShellSwap.

| Shellcode | Length | # Instruction | Source |
|---|---|---|---|
| $S_1$ | 26 Bytes | 7 | ForAllSecure |
| $S_2$ | 29 Bytes | 11 | ForAllSecure |
| $S_3$ | 22 Bytes | 12 | Shellphish |
| $S_4$ | 37 Bytes | 8 | Shellphish |
| $S_5$ | 37 Bytes | 12 | ShellSwap |

Table 4.2: The shellcode information.

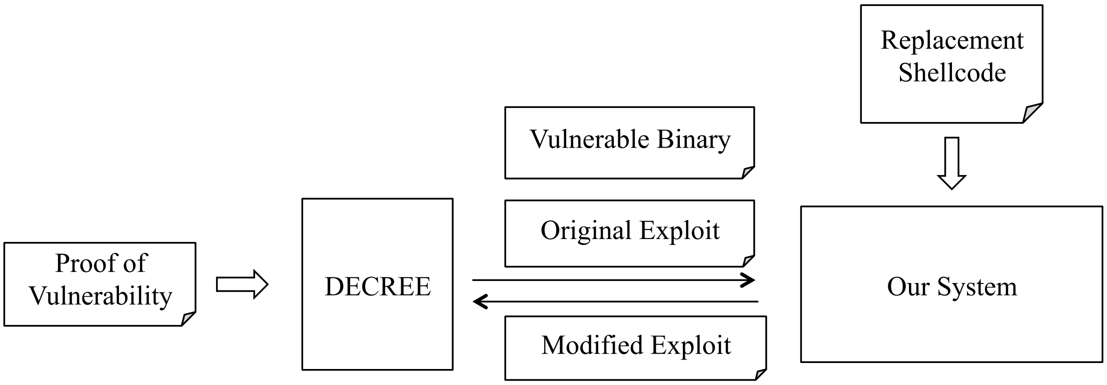


Figure 4.8: Experiment setup.

**Machines.** Our experimental setup contains two machines: one machines hosts the DARPA Experimental Cyber Research Evaluation Environment (DECREE), and the other runs ShellSwap. DECREE runs on a virtual machine built using an image provided by DARPA CGC [13, 14], which offers an isolated environment for running and testing vulnerable programs. It is assigned 1 CPU core and 1 GB of memory on a host machine with Intel Core i7 2.8 GHz. The ShellSwap machine is a standalone server with Intel Xeon E5-2630 v2 as CPU and 96 GB of memory, running Ubuntu 14.04 LTS.

**Process.** As is shown in Figure 4.8, the original exploits are pre-generated for each vulnerable binary. ShellSwap takes as input each pair of original exploit and replacement shellcode and attempts to generate a modified exploit. We verify the modified exploit against the binary in DECREE box to make sure that it works and that the replacement shellcode is executed with intended results. For testing and verification, we modified the utility script `cb-replay-pov` shipped in DECREE.

**Reference system for comparison.** To demonstrate the necessity of our approach in tackling the shellcode transplant problem, we reimplemented the shellcode placement method in the work of Cha et al. [37] in a new system on top of angr and used it as our reference system (codenamed SystemM). We simulate shellcode transplanting in SystemM by first re-triggering the exploit and then re-constraining individual symbolic blocks in memory to the replacement shellcode one by one until the modified exploit is created. If none of the symbolic memory blocks is sufficiently large to hold the replacement shellcode, or constraining every symbolic memory block to replacement shellcode leads to an unsatisfiable exploitation state (due to path conflicts), then we deem the shellcode transplanting as having failed.

| **Shellcode** | **Layout Remediation** | **Path Kneading** |
|---|---|---|
| $S_1$ | 12 | 7 |
| $S_2$ | 13 | 5 |
| $S_3$ | 14 | 5 |
| $S_4$ | 9 | 8 |
| $S_5$ | 9 | 3 |
| **Total** | 57 | 28 |

Table 4.3: The number of the generated exploits for each shellcode and each approach.

| **Shellcode** | **Length** | **# Success** | **Success Rate** |
|---|---|---|---|
| $S_3$ | 22 Bytes | 19 | 95% |
| $S_1$ | 26 Bytes | 19 | 95% |
| $S_2$ | 29 Bytes | 18 | 90% |
| $S_4$ | 37 Bytes | 17 | 85% |
| $S_5$ | 37 Bytes | 12 | 60% |

Table 4.4: Success rate for each instance of replacement shellcode, sorted by length.

### 4.5.3 Effectiveness

Table 4.1 presents the effectiveness comparison between SystemM and ShellSwap. There is a significant difference between the number of modified exploits the two systems successfully generated: SystemM successfully generated 31 exploits, whereas ShellSwap successfully generated

85 exploits. The success rate for SystemM and ShellSwap are 31% and 85%, respectively. Not surprisingly, our method generated more new exploits than previous work.

Statistics for all modified exploits successfully generated by SystemM and ShellSwap are shown in Table 4.3. ShellSwap generated 57 exploits using only Layout Remediation and 28 more by leveraging Path Kneading. For comparison, we also extended SystemM with Layout Remediation, resulting in, as expected, an additional 26 more exploits over the base SystemM implementation. Only 57% of all cases are successfully replaced with new shellcode without Path Kneading, which demonstrates the importance of conflict identification and kneading of the exploit path during shellcode replacement.

In addition, we evaluate the two-stage fallback on all 20 exploits: we replace the original shellcode in each exploit with the fallback shellcode and generate new exploits[2]. In our experiment, the two-stage fallback worked on 19 out of 20 exploits. This is because the fallback shellcode is shorter (19 bytes) than any instance of the replacement shellcode, and is thus more likely to fit into buffers under attacker controls.

Meanwhile, we observe that the success rate of shellcode transplanting varies between different instances of replacement shellcode (see Table 4.4). There is an expected negative correlation between the success rate and the length of the replacement shellcode. For example, shellcode $S_4$ and $S_5$, which are both 37 byte long, have lower success rates than other replacement shellcode that are shorter. This fits with our intuition that the longer a piece of shellcode is, the more conflicts it might produce during the shellcode transplant step, and the more difficult it will be to generate a modified exploit.

Other results are less intuitive. For instance, $S_5$ has a lower success rate than $S_4$, which is the same size. We looked into failure cases, and we found that the failure is related to the null byte in $S_5$. $S_4$ does not contain any null bytes. This conforms to the common knowledge that null bytes complicate shellcode, which is why they are generally avoided by exploit authors: since null bytes are so frequently used as string terminators, the existence of null bytes may negatively

---

[2] We do not evaluate all five instances of shellcode since any shellcode will work in the second stage.

impact the success of the exploit if data is moved around using something like `strcpy`.

### 4.5.4 Efficiency

| **Shellcode** | **Layout Remediation** | **Path Kneading** |
|---|---|---|
| $S_1$ | 18.85 | 5638.30 |
| $S_2$ | 21.05 | 10993.84 |
| $S_3$ | 20.38 | 8017.11 |
| $S_4$ | 21.01 | 7993.11 |
| $S_5$ | 17.36 | 14492.62 |
| **Average** | 19.73 | 9426.99 |

Table 4.5: Average time cost (in seconds) for each instance of replacement shellcode and each approach.

Table 4.5 shows the time cost for each instance of replacement shellcode and each approach. The average time cost for Layout Remediation is 19.73 seconds, while the average time cost for Path Kneading is 9426.99 seconds. The dramatic difference between the two is because the latter requires one or more iterations of symbolic tracing, which, as we have previously discussed, is an extremely time consuming process. We leave further performance improvement as future work, and note that there are example optimizations in related work that could be applied to this problem.

## 4.6 Discussion

ShellSwap's results open up new possibilities for the fast adaptation and analysis of software exploits. In this section, we explore the implications of these results, the limitations of the system, and the direction of our future work.

### 4.6.1 Ethical Concerns

ShellSwap raises the concern that it enables malicious attackers to quickly adapt exploits against unwitting victims on the internet. Unfortunately, such criticism can be applied to almost all security research. Similar to known techniques such as automatic exploit generation [19, 37] or automatic patch-based exploit generation [34], the merit of the ShellSwap system and its solution of the shellcode transplant problem is to show the potential abilities of attackers and to highlight the possibility that one can automatically modify exploits to tailor attacks to custom requirements. Our hope is that, by showing that this is possible, ShellSwap will motivate new research into defenses against customized exploits.

### 4.6.2 Limitation

While ShellSwap makes fundamental contributions toward the solution of the shellcode transplant problem, there is still work left to be done. Here, we discuss specific weaknesses of the system that could be addressed by future work.

**Other Types of Vulnerabilities.** Our system focuses solely on control-flow hijack vulnerabilities, and we do not address other vulnerability types, such as Information Leakage and Denial of Service (DoS). To consider these types of vulnerabilities, as well as other popular types, the shellcode transplant problem would need to be redefined, as shellcode is not utilized in exploits targeting these vulnerabilities. Thus, to generalize ShellSwap, we must first define the analogous problem in the context of a different vulnerabilities, and then discuss possible designs to solve it.

We define the analogous problem for information leakage vulnerabilities as the generation of a modified exploit that leaks a *different* piece of data (whether a memory location, a file, a variable in the program, etc.) than the original exploit does. This is a complex task to accomplish: information leakage exploits are hard to detect in the first place because monitoring the information flow through a program is not EM-enforceable in general. However, weaker variants such as taint tracking can find a smaller set of information leakage vulnerabilities. For example, evidence shows that Valgrind can detect information leakage exploits such as the Heartbleed attack [116], given test cases that trigger it (i.e., an exploit). Since, by definition, ShellSwap receives such an exploit as input, a possible method for ShellSwap to function on information leakage is to use symbolic execution to find the correlation between the exploit and the leaked information or its reference, and modify it accordingly. In this case, the memory conflicts will likely not come into play (since they are specific to placing replacement shellcode in memory), but path conflicts will still occur, and will need to be kneaded away, due to the modifications required to re-target the leak. After identifying the relation, one can come up with an exploit by solving the constraints.

We define the ricochet problem for Denial of Service vulnerabilities as the generation of a modified exploit that causes the same effect to the vulnerable program. Of course, there is little modification required – if the original exploit makes the program crash or hang at a given point, the modified exploit should have the same effect. In this case, ShellSwap is used purely as an exploit replaying system.

**Exploit Replayabilty.** ShellSwap assumes that the original exploit is deterministically replayable, in the sense that the exploit always succeeds when re-launched against the target. However, this assumption does not always hold. For instance, a vulnerable server may implement a challenge-response protocol that requires the client to send messages with a nonce that the two sides negotiated at the beginning of the session. This nonce would change when we replay the exploit, and the exploit would fail. Asymmetric encryption and sources of randomness from the environment can also manifest in such failures. To generate the modified exploit for such case, ShellSwap would have to consider an exploit as a state machine rather than a series static bytes, which would require fundamental extensions of the design.

This being said, our experiments showed that most of the exploits in our dataset *are* replayable, and our system is applicable for this majority. We intend to investigate the replaying of non-deterministic exploits in future work.

**Modern Defense Mechanisms.** Modern systems have memory protection mechanisms such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP). However, such protection mechanisms can be bypassed by properly-crafted exploits.

Our solution to the shellcode transplant problem is based on an functional original exploit, which implies that this exploit has already bypassed the required defense mechanisms. When this is the case, ShellSwap's replacement exploit often bypasses these mitigation techniques as well. For example, DEP is often bypassed through the use of Return Oriented Programming that chains pieces of code (termed *gadgets*) in a program to map an executable page (using the Linux `mmap` or DECREE `allocate` syscalls), insert shellcode into it, and execute it. Alternatively, the page containing the shellcode (for example, the program stack) can simply be marked executable by `mprotect`. For such exploits, ShellSwap bypasses DEP by reusing the original exploit's DEP bypass and replacing the final mapped shellcode with the replacement shellcode. If the replacement shellcode cannot be located at the same location as the original shellcode, the final control flow transfer of the mitigation bypass stage must be modified to point at the new location. This can be done with the constraint solver as an adaptation of the Path Kneading phase discussed in Section 4.3.

However, in the more general case of DEP bypass (for example, when a pure ROP payload is used, with no mapped shellcode), future work is required to solve the *ROP chain transplant problem*.

Bypassing ASLR is similar. One way to bypass ASLR, in the absence of DEP, is to overwrite the instruction pointer to point to `jmp *%reg` with a register `%reg` referring to a register location that currently points to the shellcode. A typical instance of the instruction is `jmp *%esp`. For the ShellSwap system, the modified exploit is able to bypass the ASLR protection if 1) the original exploit is able to bypass ASLR and 2) the beginning of the replacement shellcode is placed at the same start address as the shellcode in the original exploit (to which control flow is transferred after DEP bypass, for example). In this way, when the program dereferences a function pointer or returns a function, it will jump to the address of the start of the original shellcode, which is also the start of the replacement shellcode, and the modified exploit will succeed. Again, the final shellcode location can be modified through an adaptation of the Path Kneading phase.

More complex cases, including exploits that require an information disclosure step (to break ASLR), are currently not supported by ShellSwap. We plan to explore these in future work, and would welcome collaboration in this area.

### 4.6.3 Future Work

We plan to explore, and hope to see other researchers investigate, four main areas of future work.

First, ShellSwap can be extended to deal with encrypted, packed, or obfuscated traffic. In theory, our approach can handle these cases, because we assume knowledge the encryption key and because the decryption/decoding/deobfuscation functionality is in the original binary. However, the exploration of cases that do *not* assume knowledge of the encryption key would be interesting (albeit probably impossible in cryptographically-secure cases). A further generalization of this is the ability to successfully transplant shellcode in the presence of nondeterminism. Currently, ShellSwap cannot handle nondeterministic behavior, and some fundamental problems would need to be addressed to enable its operation on this.

Second, it would be interesting to make ShellSwap usable in an on-line capacity, where instead of modifying exploits and launching them at a later date, ShellSwap could perform the exploit live against the remote system, modifying it as appropriate based on that system's operation. Symbolic tracing is the current bottleneck of achieving this capability, but it can likely be improved by leveraging optimizations from related work [20, 92, 100]. Interestingly, the ability to function on-line would allow ShellSwap to reason about information disclosure in the course of an exploit to defeat ASLR, which is something that is not currently possible.

Third, the extension of ShellSwap to the *ROP chain transplant problem* would be an interesting future direction. Related work in the field of automatic ROP payload generation can be leveraged toward this end [99, 107].

Finally, ShellSwap can be expanded to support the generation of shellcode that is semantically equivalent to the replacement shellcode while having different contents to satisfy path constraints. Such shellcode polymorphism would increase the cases in which ShellSwap can resolve path conflicts. For example, we could consider building up a dictionary of "instruction synonyms", or creating templates to interchange instructions without changing the semantics.

## 4.7 Related Work

### 4.7.1 Automatic Exploit Generation

An exploit is valuable to attackers only when it suits attackers' specific goal. The technique of automatically generating an exploit with a piece of shellcode is called automatic exploit generation (AEG) [19, 34, 74, 99]. Those work are mostly based on dynamic symbolic execution. AEG is closely related to ShellSwap in the sense that they both take a vulnerable program and a piece of shellcode and generate a viable exploit.

Helaan et al. [63] proposed how to place shellcode in memory: scan through the memory and find symbolic memory gaps that are big enough to hold the entire piece of shellcode. For each gap, they try to put shellcode at different offsets by constraining symbolic memory bytes beginning at that offset to the actual bytes of the shellcode. This procedure continues until the shellcode is put in a memory gap or all gaps have been tried.

As we have demonstrated in our evaluation, AEG techniques are not suitable for shellcode transplanting, as they lack principled approach to diagnose and resolve conflicts imposed by replacement shellcode, and must resort to symbolic exploration. Our system makes it possible to *adapt* and *retrofit* an existing exploit to different instances of shellcode efficiently.

### 4.7.2 Intrusion Detection

In ShellSwap we detect attacks triggering software vulnerabilities and capture exploits by enforcing a set of taint-based security policies during dynamic symbolic tracing. Traditionally, taint tracking implemented on dynamic binary instrumentation frameworks (e.g. Pin [76] and Valgrind [84]) is used to detect attacks during runtime, Xu et al. [117], Autograph[70], Vigilante [41], and Bouncer [40] are all reasonable choices. While those solutions are more performant than symbolic tracing, ShellSwap cannot use them as they do not record path constraints, which are vital to our approach.

### 4.7.3 Manual Ricochet Attacks in the Wild

Ricochet attacks are widely adopted in competitive attack-defense contests today. The CTF team Shellphish has stated at DEF CON:

> *"Stealing and replaying exploits has become very popular; basically, it is the main way in which most teams attack others these days. I think that, during the last DEF CON, a majority of our flags (aka points) were coming from running stolen exploits."*

The CTF team PPP has also stated they inspected network traffic to find new vulnerabilities, which helped them score points and win DEF CON CTF in 2013 and 2014.

However, while the concept of ricochet attacks is well-known within the hacking-competition community [95, 113], it does not appear to have received much direct attention elsewhere. To the best of the knowledge, our system is the first end-to-end automatic ricochet attack generation system.

## 4.8 Conclusion

In this chapter, we introduce the automatic shellcode transplanting problem. Given a program, an exploit and a piece of shellcode, this problem asks how to automatically generate a new exploit that targets the potentially unknown vulnerability present in the program and executes the given shellcode.

We also propose ShellSwap, which is the system for automatic shellcode transplant for remote exploits. To our best knowledge, the ShellSwap system is the first automatic system that generally apply different shellcode on the exploits for unknown vulnerabilities. In our experiment, we evaluated the ShellSwap system on a combination of 20 exploits and 5 pieces of shellcode that are independently developed and different from the original exploit. Among the 100 test cases, our ShellSwap system successfully generated 85% of the exploits. Our results imply that exploit generation no longer requires delicate exploit skills. For those victims who are not familiar with exploit knowledge, they can also generate their exploits.

# Part II

# Offense and Defense Strategy

# Chapter 5

# Computing Optimal Strategy via Nash Equilibra

Automated techniques and tools for finding, exploiting and patching vulnerabilities are maturing. In order to achieve an end goal such as winning a cyber-battle, these techniques and tools must be wielded *strategically*. Currently, strategy development in cyber – even with automated tools – is done manually, and is a bottleneck in practice. In this dissertation, we apply game theory toward the augmentation of the human decision-making process.

Our work makes two novel contributions. First, previous work is limited by strong assumptions regarding the number of actors, actions, and choices in cyber-warfare. We develop a novel model of cyber-warfare that is more comprehensive than previous work, removing these limitations in the process. Second, we present an algorithm for calculating the optimal strategy of the players in our model. We show that our model is capable of finding better solutions than previous work within seconds, making computer-time strategic reasoning a reality. We also provide new insights, compared to previous models, on the impact of optimal strategies.

## 5.1 Introduction

In recent years, security researchers have pursued automated vulnerability detection and remediation techniques, attempting to scale such analyses beyond the limitations of human hackers. Eventually, automated systems will be heavily, and maybe predominantly, involved in the identification, exploitation, and repair of software vulnerabilities. This will eliminate the bottleneck that human effort represented in these areas. However, the human bottleneck (and human fallibility) will remain in the higher-level *strategy* of what to do with automatically identified vulnerabilities, automatically created exploits, and automatically generated patches.

There are many choices to make regarding the specificities of such a strategy, and these choices have real implications beyond cyber-security exercises. For example, nations have begun to make decisions on whether to disclose new software vulnerabilities (zero-day vulnerabilities) or to exploit them for gain [58, 118]. The NSA recently stated that 91% of all zero-days it discovers are disclosed, but only after a deliberation process that carefully weighs the opportunity cost from disclosing and finally forgoes using a zero-day [91]. Before disclosure, analysts at the NSA manually consider several aspects, including the potential risk to national security if the vulnerability is unpatched, the likelihood of someone else exploiting the vulnerability, and the likelihood that someone will re-discover the vulnerability [45].

In this chapter, we explore the research question of augmenting this human decision-making process with automated techniques rooted in game theory. Specifically, we attempt to identify the best strategy for the use of an identified zero-day vulnerability in a “cyber-warfare” scenario where any action may reveal information to adversaries. To do this, we create a game model where each new vulnerability is an event and “players” make strategic choices in order to optimize game outcomes. We develop our insight into optimal strategies by leveraging formal game theory methodology to create a novel approach that can calculate the best strategy for all players by computing a Nash equilibrium.

Prior work in the game theory of cyber warfare has serious limitations, placing limits on the maximum number of players, requiring perfect information awareness for all parties, or only supporting a single action on a single "event" for each player throughout the duration of the entire game. As we discuss in Section 3.6, these limitations are too restrictive for the models to be applicable to real-world cyber-warfare scenarios. Our approach addresses these limitations by developing a multi-round game-theoretic model that accounts for attack and defense in an imperfect information setting. Technically, our game model is a partial observation stochastic game (POSG) where games are played in rounds and players are uncertain about whether other players have discovered a zero-day vulnerability.

Additionally, our model supports the concept of *sequences* of player actions. Specifically, we make two new observations and add support for them to our model. First, attacks launched by a player can be observed and reverse-engineered by that player's opponents, a phenomenon we term "attack *ricocheting*" [24]. Second, patches created by a player can be reverse-engineered by their opponents to identify the vulnerability that they were meant to fix, a concept called "automated patch-based exploit generation" (APEG) in the literature [34]. This knowledge, in turn, can be used to create vulnerabilities to attack other players.

A central challenge in our work was to develop an approach for computing the Nash equilibrium of such a game. A Nash equilibrium is a strategy profile in which none of the players will have more to gain by changing the strategy. It characterizes the stable point of the game interaction in which all players are rationally playing their best responses. However, computing a Nash equilibrium is known as a Polynomial Parity Argument on Directed Graphs-complete (PPAD-complete) problem [49], which is believed to be hard. We overcome this problem in our context by taking advantage of specific characteristics of cyber-warfare games, allowing us to split the problem of computing the Nash equilibrium into two sub-problems, one of which can be converted into a Markov decision process problem and the other into a stochastic game. Our algorithm is able to compute the Nash equilibrium in polynomial time, guaranteeing the tool's applicability to real-world scenarios. Using the new model and the new algorithm, our tool finds better strategies than previous work [81, 97].

Contrary to previous work, we find that players have strategies with more utility than to attack or to disclose all the time throughout the game. Specifically, we find that in some situations, depending on various game factors, a player is better served by a patch-then-attack strategy (e.g., the NSA could disclose the vulnerability, patch their software, and then still attack) or by a pure disclose strategy (see § 5.5). These are new results not predicted by previous models. Our tool not only found the order of the actions, but also provided a concrete plan for actions over rounds, such as "patch, then attack after 2 rounds of patching".

Moreover, we observe that a previous result of prior work in the area – the concept that it is always optimal for at least one player to attack – does not stand in our expanded model (see § 5.5). We demonstrated this by showing an example where the optimal strategy for both players is to disclose and patch. We also observe that a player must ricochet *and* patch fast enough in order to prevent his opponent from attacking and forcing the vulnerability disclosure. If a player is only able to either ricochet *or* patch fast enough, they might still get attacked.

The optimal strategies derived from our models have real-world consequences – as a case study of our work, we apply our model to a recent fully-automated cyber security contest run by DARPA, the Cyber Grand Challenge, which had 4.25 million dollars in prizes. Our study shows that an adoption of our model by the third-place team, team Shellphish, would have heavily improved their final standing. The specific strategy picked in the approach to cyber warfare *matters*, and these choices have real-world consequences. In the CGC case, the consequence was a difference of prize winnings, but in the real world (the challenges of which the CGC was designed to mirror [47]), the difference could be more fundamental.

Overall, our work makes the following contributions:

- We develop a novel model of cyber-warfare that is more comprehensive than previous work because it 1) considers strategies as a sequence of actions over time, 2) addresses players' uncertainty about their opponents, and 3) accounts for more offensive and defensive techniques that can be employed for cyber-warfare, such as ricocheting and APEG (Section 5.2).

- We present an algorithm for calculating the optimal strategy of the players in our model. The original model is a POSG game, which, in general, is intractable to solve [79]. We take advantage of the structure of cyber-warfare and propose a novel approach for finding the Nash equilibrium (Section 5.3).
- We show that our model is capable of finding better solutions than previous work within seconds. We also provide new insights, compared to previous models, on the impact of optimal strategies. We demonstrate that optimal strategies in a cyber-warfare are more complex than previous conclusions [81, 97], and one must take into consideration the unique aspect of cyber-warfare (versus physical war), i.e., that exploits can be generated by the ricochet and APEG techniques. This insight leads to new equilibriums *not* predicted by previous work [81, 97] (Section 5.5).

## 5.2 Problem Statement and Goals

In this section, we formally state the cyber-warfare game. We will describe the problem setup, lay out our assumptions, present the formalized POSG model and finally clarify the goal of our work.

### 5.2.1 Problem Setup

The cyber-warfare game needs to be general and compatible with known cyber-warfare events such as the Stuxnet event. One requirement is that the cyber-warfare game model must support players with comprehensive techniques, rather than the simple choices that prior models allow. Figure 5.1 shows the workflow of the player in our model. There are three ways for a player to learn of a vulnerability. A player may detect an attack, receive the disclosure from other players, or discover the vulnerability by himself. After a player learns a vulnerability, the strategy

| Parameter | Definition |
|---|---|
| $p_i(t)$ | The probability distribution over time that player $i$ discovers a vulnerability at round $t$. |
| $q_i(t)$ | The probability to launch a ricochet attack with exploits that player $i$ received in the previous round. |
| $h_i(t)$ | The ratio of the amount of patched vulnerable resources over the total amount of vulnerable resources by $t$ rounds *after* the vulnerability is disclosed. |
| $\delta_i$ | The number of rounds required by player $i$ to generate a patch-based exploit after a vulnerability and the corresponding patch are disclosed. |
| $u_i(t)$ | The dynamic utility that player $i$ gains by attacking his opponents at round $t$. |

Table 5.1: Parameters for player $i$.

generator will compute the strategy for the vulnerability.

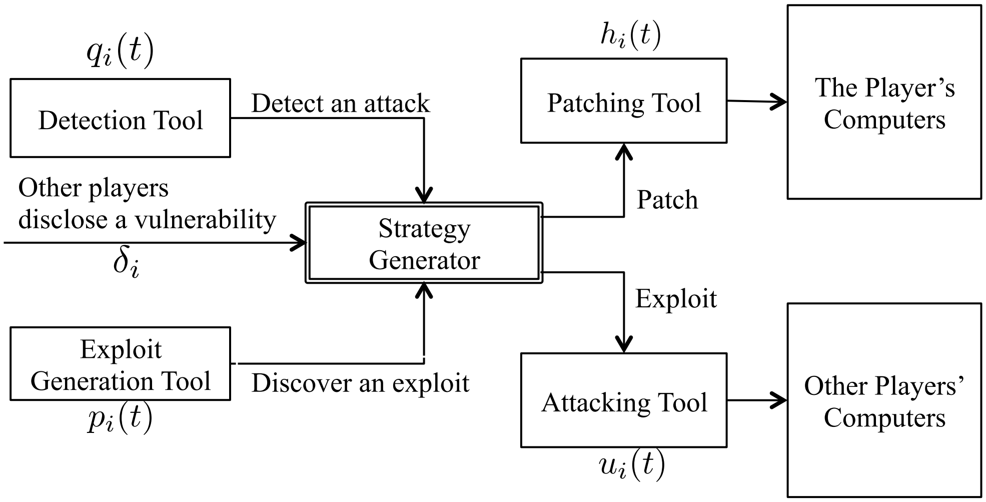


Figure 5.1: The workflow of the player in the cyber-warfare game.

**Players.** All players are participating in a networked scenario. They are capable of finding new vulnerabilities, monitoring their own connections and ricocheting an attack, patching after vulnerability disclosure, and generating patch-based exploits. Players may have different levels of skills, which are characterized using the parameters listed in Table 5.1.

These parameters are a substantial component of our model. For player $i$, the vulnerability discovery skill is denoted by $p_i(t)$, which is a function of probability that the player discovers a zero-day vulnerability distributed over rounds. Their level of ricochet ability is characterized by parameter $q_i(t)$, which is the probability of launching a ricochet attack with exploits they recovered from the traffic received in the previous round.

A player's patching skill is represented by function $h_i(t)$. $h_i(t)$ is the ratio of the amount of patched vulnerable resources over the total amount of vulnerable resources by $t$ rounds *after* the vulnerability is disclosed. In the real world, while patching a single computer might take only minutes, patching all vulnerable resources (depending on the organization, containing thousands of instances) might take days to months [27]. While one player is patching, other players could possibly attack, and any vulnerable resources that have not been patched will suffer from the attack.

The last parameter, $\delta_i$, describes player $i$'s level of APEG skills, which is the number of rounds required by the player to generate a patch-based exploit after a vulnerability and the corresponding patch are disclosed. Finally, the attacking utility, denoted as $u_i(t)$, encodes the *dynamic* utility that player $i$ gains by attacking his opponents at round $t$ before the patch is released.

**Player States and Actions.** Each player $i$ has a state denoted by $\theta_i$ in each round, where $\theta_i \in \Theta_i = \{\neg\mathrm{D}, \mathrm{D}\}$. $\neg$D refers to the situation in which a player has not yet learned of a zero-day, while D refers to the situation in which a player knows the vulnerability, either by actively finding the vulnerability and developing an exploit or by passively "stealing" the exploit from an attack or a patch.

In each round, players choose one of the following actions: $\{$ATTACK, PATCH, NOP (no action), STOCKPILE$\}$, where the semantics of ATTACK, PATCH, and NOP have their literal meaning, and STOCKPILE means holding a zero-day vulnerability for future use.

Players are limited in their actions by their state. This is acceptable for stochastic games and does not impact the difficulty or insights of the model [73, §6.3.1]. In particular, while a player in state ¬D can only act NOP, a player in state D can choose an action among ATTACK, PATCH and STOCKPILE before the patch is released, and between ATTACK and NOP after the release, depending on their skill at detecting attacks or APEG.

### 5.2.2 Game Factors & Assumptions

Our model considers the game within the scope of the following factors and assumptions:

- We do not distinguish between a player discovering the vulnerability and knowing about how to exploit it in our game. This is because in many cases, when a nation acquires a zero-day vulnerability, the nation also learns about the zero-day exploit. For example, nations acquire zero-day vulnerabilities by purchasing zero-day exploits from vulnerability markets [58, 60, 87]. We acknowledge that discovering a vulnerability and creating an exploit are conceptually different, but we leave the separation of those two as future work.
- We do not distinguish between a player disclosing a vulnerability and releasing a patch. The known patching mechanisms such as Microsoft update make secret patching unlikely to happen in the real world, and in most cases, the disclosure of a vulnerability comes with a patch or a workaround.
- We assume that players are monitoring their systems, and may probabilistically detect an attack. We also assume they may be able to then *ricochet* the exploit to other players. We note that the detection may come through monitoring of the network (in the case of network attacks), or other measures such as honeypots, dynamic analysis of suspicious inputs, etc. For example, Vigilante [41] detects exploits and creates a self-certifying alert (SCA), which is essentially a replayable exploit. We note that such attacks may be detected over a network (e.g., in CTF competitions, these are called reflection attacks [95]) or via dynamic analysis, as with Vigilante.
- We assume that once a patch reveals information about the vulnerability, other players can use the patch to create an exploit. For example, Brumley et al. shows this can be done automatically

in some instances [34]. We note that patch-based exploit generation is useful because the resulting exploit can be used before the patch is applied on all vulnerable systems.

### 5.2.3 Formalization

We formalize the cyber security game as a $n$-player zero-sum partial observation stochastic game:

$$POSG = \langle \mathcal{N}^P, \mathcal{A}^P, \Theta^P, \Phi^P, R^P \rangle.$$

In this chapter, we focus on 2 players ($|\mathcal{N}^P| = 2$) called player 1 and player 2.

**Game State.** The complete game state $\Theta^P$ is defined as $\mathcal{T} \times \mathcal{R} \times \Theta_1 \times \Theta_2$, where $\mathcal{T}$ is the round number, $\mathcal{R}$ is the specific round when the patch is released, and $\Theta_i$ is the set of player $i$'s states ($\Theta_i = \{\neg\mathrm{D}, \mathrm{D}\}$). The round of releasing a patch is $\emptyset$ before a patch is released. The patch release time is needed because it is a public indicator of the discovery of the vulnerability. We use this to bound uncertainty, since after a patch is released every player has the potential (and eventual) understanding of the vulnerability.

**State Transition.** In each round, the game is in a concrete state, but the players have incomplete information about that state. The players make an observation and then choose an action. The chosen actions transition the game to a new state. The transition function is public for both players. Transitions in a game may be probabilistic. We denote the probability transition function over game states by $\Phi^P : \Theta^P \times \mathcal{A}_1^P \times \mathcal{A}_2^P \to \Delta(\Theta^P)$ and show these transitions in Figure 5.2. This divides the game state into five categories:

**a.** Neither player has discovered a vulnerability (Figure 5.2a, $\langle t, \emptyset, \neg\mathrm{D}, \neg\mathrm{D} \rangle$).

The available action for each player is NOP, and the probability that player $i$ discovers a vulnerability in the current round is $p_i(t)$. Since players discover vulnerabilities independently, the joint probability of player 1 in state $\theta_1$ and player 2 in $\theta_2$ is equal to the product

$$\begin{tikzcd}
 & & {} \arrow[r, "(1-p_1(t))(1-p_2(t))"] & \langle t+1,\emptyset,\neg\mathrm{D},\neg\mathrm{D}\rangle \\
\langle t,\emptyset,\neg\mathrm{D},\neg\mathrm{D}\rangle & \langle\mathrm{NOP},\mathrm{NOP}\rangle & {} \arrow[r, "p_1(t)(1-p_2(t))"] & \langle t+1,\emptyset,\mathrm{D},\neg\mathrm{D}\rangle \\
 & & {} \arrow[r, "(1-p_1(t))p_2(t)"] & \langle t+1,\emptyset,\neg\mathrm{D},\mathrm{D}\rangle \\
 & & {} \arrow[r, "p_1(t)p_2(t)"] & \langle t+1,\emptyset,\mathrm{D},\mathrm{D}\rangle
\end{tikzcd}$$

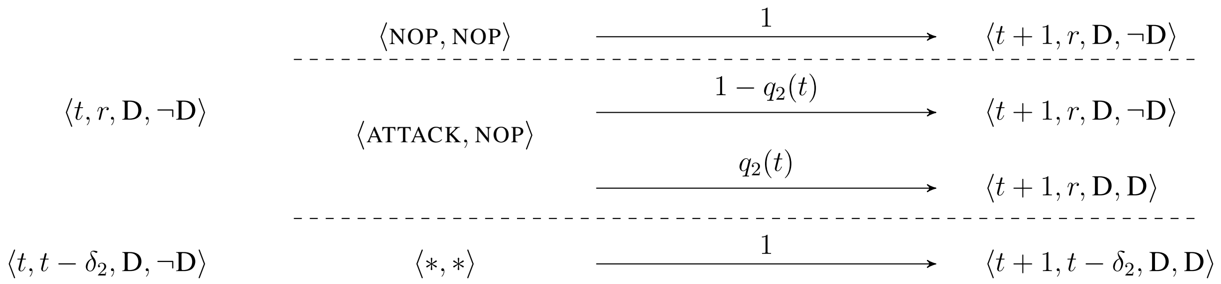
(a) Neither player has discovered a vulnerability.

$$\begin{tikzcd}
 & & {} \arrow[r, "(1-p_1(t))(1-q_1(t))"] & \langle t+1,\emptyset,\neg\mathrm{D},\mathrm{D}\rangle \\
 & \langle\mathrm{NOP},\mathrm{ATTACK}\rangle & {} \arrow[r, "q_1(t)+p_1(t)(1-q_1(t))"] & \langle t+1,\emptyset,\mathrm{D},\mathrm{D}\rangle \\
\langle t,\emptyset,\neg\mathrm{D},\mathrm{D}\rangle & & {} \arrow[r, "p_1(t)"] & \langle t+1,\emptyset,\mathrm{D},\mathrm{D}\rangle \\
 & \langle\mathrm{NOP},\mathrm{STOCKPILE}\rangle & {} \arrow[r, "(1-p_1(t))"] & \langle t+1,\emptyset,\neg\mathrm{D},\mathrm{D}\rangle \\
 & \langle\mathrm{NOP},\mathrm{PATCH}\rangle & {} \arrow[r, "1"] & \langle t+1,t,\neg\mathrm{D},\mathrm{D}\rangle
\end{tikzcd}$$

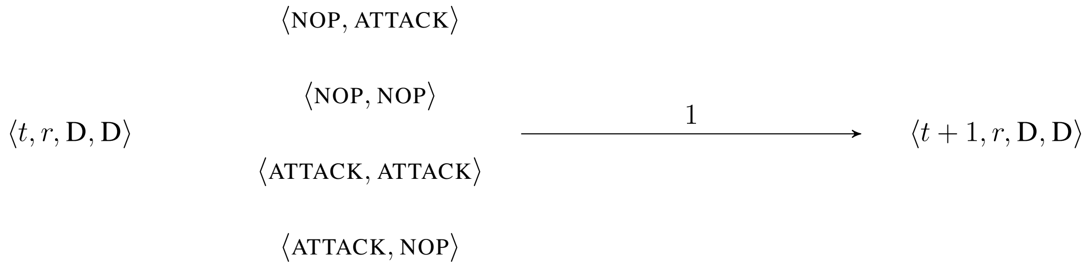
(b) One player (player 2) has discovered a vulnerability.

$$\begin{tikzcd}
 & \langle\mathrm{STOCKPILE},\mathrm{ATTACK}\rangle & & \\
 & \langle\mathrm{STOCKPILE},\mathrm{STOCKPILE}\rangle & {} \arrow[r, "1"] & \langle t+1,\emptyset,\mathrm{D},\mathrm{D}\rangle \\
\langle t,\emptyset,\mathrm{D},\mathrm{D}\rangle & \langle\mathrm{ATTACK},\mathrm{ATTACK}\rangle & & \\
 & \langle\mathrm{ATTACK},\mathrm{STOCKPILE}\rangle & & \\
 & \langle *,\mathrm{PATCH}\rangle & & \\
 & & {} \arrow[r, "1"] & \langle t+1,t,\mathrm{D},\mathrm{D}\rangle \\
 & \langle\mathrm{PATCH},*\rangle & &
\end{tikzcd}$$

(c) Both players have discovered and withheld the vulnerability.


$\langle t, r, \mathrm{D}, \neg\mathrm{D} \rangle$
$\langle \text{NOP}, \text{NOP} \rangle$
$1$
$\langle t+1, r, \mathrm{D}, \neg\mathrm{D} \rangle$
$\langle \text{ATTACK}, \text{NOP} \rangle$
$1 - q_2(t)$
$\langle t+1, r, \mathrm{D}, \neg\mathrm{D} \rangle$
$q_2(t)$
$\langle t+1, r, \mathrm{D}, \mathrm{D} \rangle$
$\langle t, t-\delta_2, \mathrm{D}, \neg\mathrm{D} \rangle$
$\langle *, * \rangle$
$1$
$\langle t+1, t-\delta_2, \mathrm{D}, \mathrm{D} \rangle$

(d) One player (player 1) has discovered and disclosed a vulnerability, while the other player has not discovered the vulnerability.


$\langle t, r, \mathrm{D}, \mathrm{D} \rangle$
$\langle \text{NOP}, \text{ATTACK} \rangle$
$\langle \text{NOP}, \text{NOP} \rangle$
$\langle \text{ATTACK}, \text{ATTACK} \rangle$
$\langle \text{ATTACK}, \text{NOP} \rangle$
$1$
$\langle t+1, r, \mathrm{D}, \mathrm{D} \rangle$

(e) Both players have discovered the vulnerability, and the vulnerability has been disclosed.

Figure 5.2: The transitions of game states. For each sub-figure, the left-hand side is the state of the current round and the right-hand side is the state of the next round.

each player is in his respective state.

**b.** Only one player has discovered a vulnerability (Figure 5.2b, $\langle t, \emptyset, \neg\mathrm{D}, \mathrm{D}\rangle$ or $\langle t, \emptyset, \mathrm{D}, \neg\mathrm{D}\rangle$).

Suppose player 2 has the exploit, then player 2 has three possible actions while player 1 has one. If player 2 chooses to ATTACK, the probability that player 1 transits to state D is equal to the joint probability of finding the vulnerability by himself, and that detecting player 2's attack. If player 2 chooses to STOCKPILE, the probability that player 1 will be in state D in the next round is equal to the probability that he independently discovers the vulnerability. If player 2 chooses to PATCH, player states remain unchanged and the patch releasing round will be updated to $t$.

**c.** Both players have discovered the vulnerability and they withhold it (Figure 5.2c, $\langle t, \emptyset, \mathrm{D}, \mathrm{D}\rangle$).

If neither player releases a patch, the states and the patch-releasing round remain the same. Otherwise, the game will transition to $\langle t+1, t, \mathrm{D}, \mathrm{D}\rangle$.

**d.** One player has disclosed the vulnerability, while the other player has not discovered it (Figure 5.2d, $\langle t, r, \mathrm{D}, \neg\mathrm{D}\rangle$ or $\langle t, r, \neg\mathrm{D}, \mathrm{D}\rangle$).

Suppose player 1 releases the patch at round $r$, then player 2 will generate an exploit based on this patch in $\delta_2$ rounds. If player 1 chooses to NOP during those rounds, then player 2 will keep developing the patch-based exploit until the $(r + \delta_2)$-th round. Otherwise, if player 1 chooses to ATTACK, then player 2 will detect the attack with probability $q_2(t)$ and transition to state D in the next round after doing so.

**e.** Both players have discovered the vulnerability, and the vulnerability has been disclosed (Figure 5.2e, $\langle t, r, \mathrm{D}, \mathrm{D}\rangle$).

In this case, player states and the patch-releasing round remains unchanged.

**Utility.** Players' utility for each round is calculated according to the reward function for one round $R^P : \mathcal{A}_1^P \times \mathcal{A}_2^P \rightarrow \mathbb{R}$. This function is public, but the actual utility per round is secret

| | | NOT DISCOVER | DISCOVER | | |
|---|---|---|---|---|---|
| | | NOP | ATTACK | STOCKPILE | PATCH |
| NOT DISCOVER | NOP | 0 | $-u_2(t)$ | 0 | 0 |
| DISCOVER | ATTACK | $u_1(t)$ | $u_1(t) - u_2(t)$ | $u_1(t)$ | 0 |
| | STOCKPILE | 0 | $-u_2(t)$ | 0 | 0 |
| | PATCH | 0 | 0 | 0 | 0 |

Table 5.2: The reward matrix at round $t$ before patch is released. The table shows the reward of player 1 (the row player). The reward of player 2 (the column player) is the negative value of that of player 1.

| | | NOT DISCOVER | DISCOVER | |
|---|---|---|---|---|
| | | NOP | ATTACK | NOP |
| DISCOVER | NOP | 0 | $-u_2(t)h_1(t-r)$ | 0 |
| | ATTACK | $u_1(t)h_2(t-r)$ | $u_1(t)h_2(t-r) - u_2(t)h_1(t-r)$ | $u_1(t)h_2(t-r)$ |
| NOT DISCOVER | NOP | 0 | $-u_2(t)h_1(t-r)$ | 0 |

Table 5.3: The reward matrix at round $t$ after patch is released. The table shows the reward of player 1 (the row player). The reward of player 2 (the column player) is the negative value of that of player 1.

to players because players do not always know the action of the other players. We assume that the amount of utility that a player gains is equal to the amount that the other player loses, which makes our game zero-sum.

The reward function is calculated using the attacking utility functions $u_{1,2}(t)$ and the patching portion functions $h_{1,2}(t)$. We define the reward function before and after patching separately, which are shown as Table 5.2 and Table 5.3. In both tables, player 1 is the row player and player 2 is the column player.

### 5.2.4 Goals

We focus on calculating the pure strategy Nash equilibrium for our cyber-warfare model. However, computing the Nash equilibrium of a general POSG remains open even for a two-player game, due to *nested belief* [79, 119]. This means that players are concerned about not only the game state, but also the other player's belief over the game state. The players' 0-level beliefs are represented as probabilities over the game state. Based on 0-level beliefs, players must meta-reason about the beliefs that players hold about others' beliefs. These levels of meta-reasoning – called *nested beliefs* – can keep going indefinitely to infinite levels. If a player stops at a limited level of nested belief, the other players can reason about further levels of nested beliefs and change the result of the game. A player must include the infinite nested belief as part of their utility calculation when determining an optimal strategy.

## 5.3 Finding Equilibriums

Although the POSG model, as an incomplete information game, characterizes the uncertainty inherent in cyber-warfare, computing the equilibriums of a general POSG remains open. However, we discover three insights, specific to cyber-warfare, that help us reduce the complexity of the game and calculate the Nash equilibrium for our cyber-warfare game model.

First, if player $i$ releases a patch, then all players subsequently know player $i$ has found the vulnerability. We use this to split the cyber war game into two phases: before disclosure and after disclosure.

Second, players can probabilistically bound how likely another player is to discover a vulnerability based upon their skill level. This is because the probability is inferred based on players' attributes, such as the discovery probability, ricochet probability, and those attributes are public to all players.

Finally, although players are uncertain about the state of the other players (which they represent as a probability distribution of player states), they know the probability of their opponents being in a state given the public information of the opponents, such as the vulnerability discovery probability (e.g., based upon prior zero-day battles) and the ricochet probability.

Based on the above insights, we convert the POSG model to a stochastic game model by encoding the belief of each player into the game state. In our game, the belief of a player is the probability that the player thinks the other player has found the vulnerability. We can compute the Nash equilibrium for the converted stochastic game by dynamic programming. We will also discuss the observation of players' strategy after vulnerability disclosure.

### 5.3.1 The Stochastic Game

As our assumption that a player's belief about the state of opponent players can be estimated from the globally-known player properties, the POSG model reduces to a much more tractable stochastic model in the pre-disclosure phase. We define the stochastic game (SG) model

$$SG = \langle \mathcal{N}^S, \mathcal{A}^S, \Theta^S, \Phi^S, R^S \rangle$$

We retain the definition of players in POSG, $\mathcal{N}^P = \mathcal{N}^S = \{\text{player } 1, \text{player } 2\}$.

**Player Actions.** The player action in SG is defined as a combination of player actions under different player states. For example, if player $i$ plays ATTACK in state D and NOP in state ¬D, the corresponding action in the SG model is $\{\text{D} : \text{ATTACK}, \neg\text{D} : \text{NOP}\}$. For each player action $a_i$, we will use $a_i[\text{D}]$ and $a_i[\neg\text{D}]$ to denote the action in state D and ¬D, respectively.

**Game State.** The game state $\Theta^S$ in the SG model is defined as $\Theta^S = \mathcal{T} \times \mathcal{R} \times \mathbb{R} \times \mathbb{R}$. Besides the current round number $\mathcal{T}$ and the patch releasing round number $\mathcal{R}$, a game state includes the beliefs of the two players about each other, which is the probability that a player has discovered a vulnerability from the other player's perspective, $b_i \in [0, 1]$. A game state $\theta^S \in \Theta^S$ can be represented as $\theta^S = \langle t, r, b_1, b_2 \rangle$, in which player 2 thinks the probability that player 1 has discovered the vulnerability is $b_1$, and player 1 thinks the probability that player 2 has discovered the vulnerability is $b_2$.

Unlike the POSG model, the game states in the SG model include the uncertainty of a player about the other player's state. In each round of the game, players know their own states; although they do not know the other player's state, they infer the likelihood of the other player's state based on the other player's parameters. In addition, a player also knows the other player's beliefs about the game state because the player also knows the parameters of himself. Therefore, we are able to convert to the SG model under the structure of the game states above.

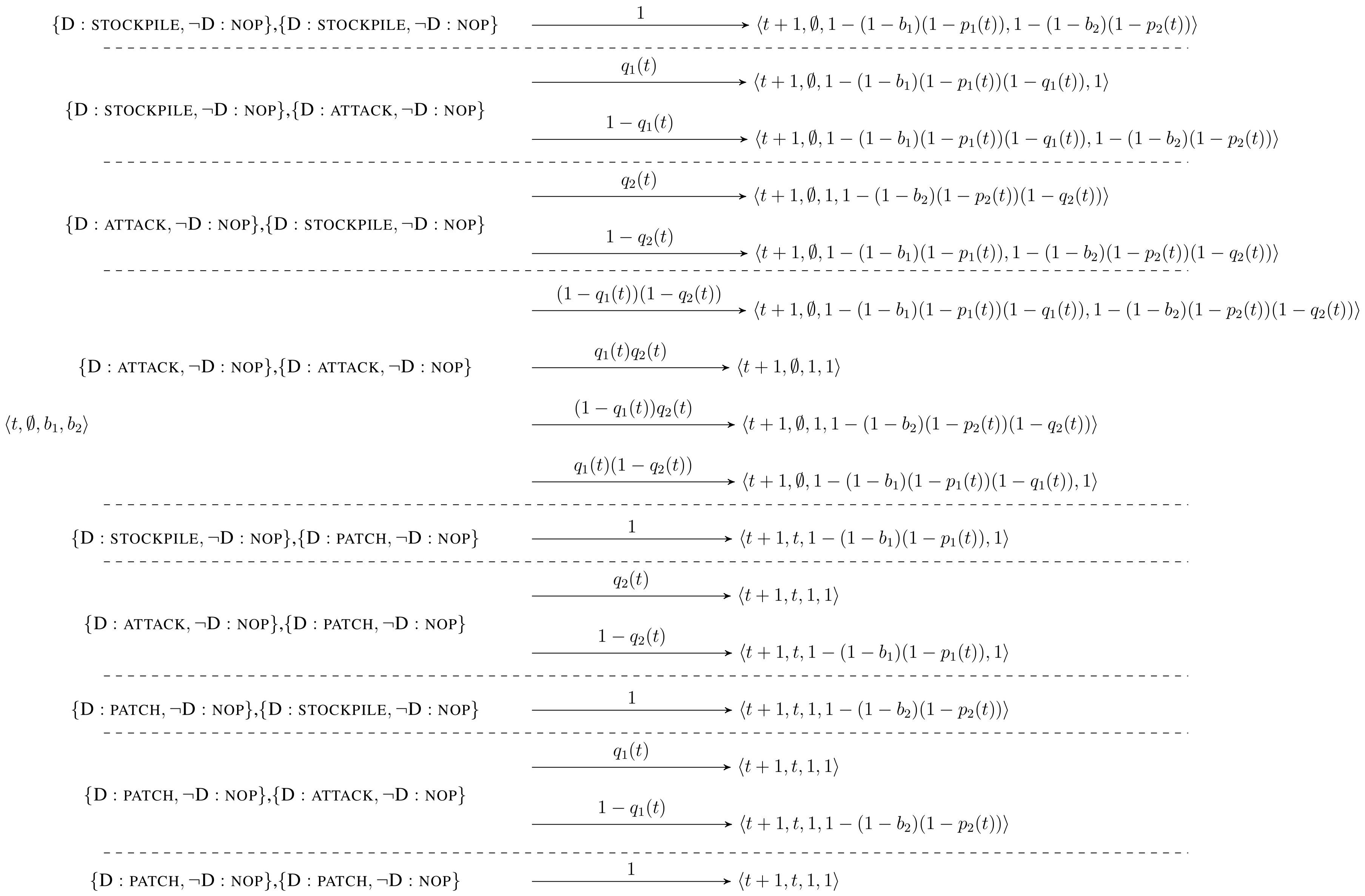
**State Transition.** We define the state transition function of the SG model as $\Phi^S : \Theta^S \times \mathcal{A}_1^S \times \mathcal{A}_2^S \to \Delta(\Theta^S)$. We represent the probability that a game transitions to $\theta^S$ using $\Phi^S(\cdot)[\theta^S]$. The transition between the game states is shown in Figure 5.3. The game states are divided by the time before and after vulnerability disclosure, because the actions and information available to players are different between the two phases.

**Before Disclosure (Figure 5.3a).** Suppose the game is in state $\langle t, \emptyset, b_1, b_2 \rangle$. If neither player acts ATTACK, the probability that player $i$ discovers the exploits at the current round is $p_i(t)$ and the probability that player $i$ discovers the exploit by the current round is $1-(1-b_i)(1-p_i(t))$. The game transits to state $\langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t)), 1-(1-b_2)(1-p_2(t)) \rangle$.

If a player chooses to ATTACK, the probability that their opponent will acquire the exploit in the current round is the joint probability that the opponent discovers the vulnerability by himself and that he detects the exploit. Meanwhile, if the opponent detects the exploit, they will be certain that the attacker has the exploit. For example, if player 1 's action is $\{\text{D} : \text{ATTACK}, \neg\text{D} : \text{NOP}\}$ while player 2 's action is $\{\text{D} : \text{STOCKPILE}, \neg\text{D} : \text{NOP}\}$, the game will transition to $\langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t)), 1-(1-b_2)(1-p_2(t))(1-q_2(t)) \rangle$ with the probability of $1-q_2(t)$ and $\langle t+1, \emptyset, 1, 1-(1-b_2)(1-p_2(t))(1-q_2(t)) \rangle$ with the probability of $q_2(t)$.

Similarly, if both players act ATTACK in state D, the game will transition to one of four possibilities. If neither player detects the exploit, the game state will be $\langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t))(1-q_1(t)), 1-(1-b_2)(1-p_2(t))(1-q_2(t)) \rangle$. If player 1 detects the exploit while player 2 does not, the game state will be $\langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t))(1-q_1(t)), 1 \rangle$. If player 2 detects the exploit while player 1 does not, the game state will be $\langle t+1, \emptyset, 1, 1-(1-b_2)(1-p_2(t))(1-q_2(t)) \rangle$. Finally, if both players detect the exploit, the game state will be $\langle t+1, \emptyset, 1, 1 \rangle$.

If one player acts PATCH, both players will patch immediately. If player 1 releases a patch,



$\langle t, \emptyset, b_1, b_2 \rangle$
$\{\mathrm{D}: \text{STOCKPILE}, \neg\mathrm{D}: \text{NOP}\},\{\mathrm{D}: \text{STOCKPILE}, \neg\mathrm{D}: \text{NOP}\}$
$\xrightarrow{1} \langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t)), 1-(1-b_2)(1-p_2(t)) \rangle$
$\{\mathrm{D}: \text{STOCKPILE}, \neg\mathrm{D}: \text{NOP}\},\{\mathrm{D}: \text{ATTACK}, \neg\mathrm{D}: \text{NOP}\}$
$\xrightarrow{q_1(t)} \langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t))(1-q_1(t)), 1 \rangle$
$\xrightarrow{1-q_1(t)} \langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t))(1-q_1(t)), 1-(1-b_2)(1-p_2(t)) \rangle$
$\{\mathrm{D}: \text{ATTACK}, \neg\mathrm{D}: \text{NOP}\},\{\mathrm{D}: \text{STOCKPILE}, \neg\mathrm{D}: \text{NOP}\}$
$\xrightarrow{q_2(t)} \langle t+1, \emptyset, 1, 1-(1-b_2)(1-p_2(t))(1-q_2(t)) \rangle$
$\xrightarrow{1-q_2(t)} \langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t)), 1-(1-b_2)(1-p_2(t))(1-q_2(t)) \rangle$
$\{\mathrm{D}: \text{ATTACK}, \neg\mathrm{D}: \text{NOP}\},\{\mathrm{D}: \text{ATTACK}, \neg\mathrm{D}: \text{NOP}\}$
$\xrightarrow{(1-q_1(t))(1-q_2(t))} \langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t))(1-q_1(t)), 1-(1-b_2)(1-p_2(t))(1-q_2(t)) \rangle$
$\xrightarrow{q_1(t)q_2(t)} \langle t+1, \emptyset, 1, 1 \rangle$
$\xrightarrow{(1-q_1(t))q_2(t)} \langle t+1, \emptyset, 1, 1-(1-b_2)(1-p_2(t))(1-q_2(t)) \rangle$
$\xrightarrow{q_1(t)(1-q_2(t))} \langle t+1, \emptyset, 1-(1-b_1)(1-p_1(t))(1-q_1(t)), 1 \rangle$
$\{\mathrm{D}: \text{STOCKPILE}, \neg\mathrm{D}: \text{NOP}\},\{\mathrm{D}: \text{PATCH}, \neg\mathrm{D}: \text{NOP}\}$
$\xrightarrow{1} \langle t+1, t, 1-(1-b_1)(1-p_1(t)), 1 \rangle$
$\{\mathrm{D}: \text{ATTACK}, \neg\mathrm{D}: \text{NOP}\},\{\mathrm{D}: \text{PATCH}, \neg\mathrm{D}: \text{NOP}\}$
$\xrightarrow{q_2(t)} \langle t+1, t, 1, 1 \rangle$
$\xrightarrow{1-q_2(t)} \langle t+1, t, 1-(1-b_1)(1-p_1(t)), 1 \rangle$
$\{\mathrm{D}: \text{PATCH}, \neg\mathrm{D}: \text{NOP}\},\{\mathrm{D}: \text{STOCKPILE}, \neg\mathrm{D}: \text{NOP}\}$
$\xrightarrow{1} \langle t+1, t, 1, 1-(1-b_2)(1-p_2(t)) \rangle$
$\{\mathrm{D}: \text{PATCH}, \neg\mathrm{D}: \text{NOP}\},\{\mathrm{D}: \text{ATTACK}, \neg\mathrm{D}: \text{NOP}\}$
$\xrightarrow{q_1(t)} \langle t+1, t, 1, 1 \rangle$
$\xrightarrow{1-q_1(t)} \langle t+1, t, 1, 1-(1-b_2)(1-p_2(t)) \rangle$
$\{\mathrm{D}: \text{PATCH}, \neg\mathrm{D}: \text{NOP}\},\{\mathrm{D}: \text{PATCH}, \neg\mathrm{D}: \text{NOP}\}$
$\xrightarrow{1} \langle t+1, t, 1, 1 \rangle$

(a) Before disclosing a vulnerability.

$$\langle t, r, 1, b_2\rangle$$

$$\{\text{D} : \textsc{stockpile}\},\{\text{D} : \textsc{stockpile}, \neg\text{D} : \textsc{nop}\} \xrightarrow{1} \langle t+1, r, 1, b_2\rangle$$

$$\{\text{D} : \textsc{stockpile}\},\{\text{D} : \textsc{attack}, \neg\text{D} : \textsc{nop}\} \begin{array}{l} \xrightarrow{q_1(t)} \langle t+1, r, 1, 1\rangle \\ \xrightarrow{1-q_1(t)} \langle t+1, r, 1, b_2\rangle \end{array}$$

$$\{\text{D} : \textsc{attack}\},\{\text{D} : \textsc{stockpile}, \neg\text{D} : \textsc{nop}\} \xrightarrow{1} \langle t+1, r, 1, b_2 + (1-b_2)q_2(t)\rangle$$

$$\{\text{D} : \textsc{attack}\},\{\text{D} : \textsc{attack}, \neg\text{D} : \textsc{nop}\} \begin{array}{l} \xrightarrow{q_1(t)} \langle t+1, r, 1, 1\rangle \\ \xrightarrow{1-q_1(t)} \langle t+1, r, 1, b_2 + (1-b_2)q_2(t)\rangle \end{array}$$

$$\langle t, t-\delta_2, 1, b_2\rangle$$

$$\left.\begin{array}{c} \{\text{D} : \textsc{attack}\},\{\text{D} : \textsc{attack}, \neg\text{D} : \textsc{nop}\} \\ \{\text{D} : \textsc{attack}\},\{\text{D} : \textsc{stockpile}, \neg\text{D} : \textsc{nop}\} \\ \{\text{D} : \textsc{stockpile}\},\{\text{D} : \textsc{attack}, \neg\text{D} : \textsc{nop}\} \\ \{\text{D} : \textsc{stockpile}\},\{\text{D} : \textsc{stockpile}, \neg\text{D} : \textsc{nop}\} \end{array}\right. \xrightarrow{1} \langle t+1, t-\delta_2, 1, 1\rangle$$

(b) After disclosing a vulnerability. Suppose player 1 discloses a vulnerability.

Figure 5.3: The transition of game states in the stochastic game. The left-hand sides of the arrows are game states with possible strategies. The right-hand sides of the arrows are the possible game states after transition.

the game will transition to $\langle t+1, t, 1, b_2\rangle$, as everyone is certain that player 1 has the exploit. If player 2 releases a patch, the game will transit to $\langle t+1, t, b_1, 1\rangle$ and if both player release a patch, the game will transition to $\langle t+1, t, 1, 1\rangle$.

**After Disclosure (Figure 5.3b).** After disclosure, both players will know the vulnerability so they will stop searching for it. Also, the player disclosing a vulnerability is public so both players know that the player is in state D. Suppose player 1 discloses a vulnerability in round $r$. In response, player 2 starts APEG and will generate the exploit by round $r+\delta_2$. Meanwhile, player 2 still has the chance to ricochet attacks if player 1 attacks. Therefore, the belief of player 2 's possession of the exploit will increase if player 1 attacks in the previous round. authnotetifSeaGreenit can expand.m

**Utility.** We calculate players' utility by the single-round reward function $R^S : \Theta^S \times \mathcal{A}_1^S \times \mathcal{A}_2^S \to \mathbb{R}$. Given a game state and players actions, the single-round reward is equal to the expected reward over player states.

Given player $i$ and a player state $\theta_i$, the probability that the player is in state $\theta_i$ when the SG game state is $\theta^S = \langle t, r, b_1, b_2\rangle$, which is denoted by $P(\theta^S, \theta_i)$, is equal to

$$P(\theta^S, \theta_i) = P(\langle t, r, b_1, b_2\rangle, \theta_i) = \begin{cases} b_i & \text{if } \theta_i = \neg\text{D} \\ 1 - b_i & \text{if } \theta_i = \text{D} \end{cases}$$

Recall the reward function for the POSG model $\mathcal{R}^P : \mathcal{A}_1^P \times \mathcal{A}_2^P \to \mathbb{R}$ takes as input the players' actual actions and produces as output the utility of one player (because the utility of the other is the negative value for zero-sum game). We calculate the reward for the SG model using $\mathcal{R}^P$. In specific, we have

$$R^S(\theta^S, a_1^S, a_2^S) = \sum_{\theta_1} \sum_{\theta_2} P(\theta^S, \theta_1) P(\theta^S, \theta_2) R^P(a_1^S[\theta_1], a_2^S[\theta_2])$$

### 5.3.2 Compute the Nash Equilibrium

A Nash equilibrium is a strategy profile where neither player has more to gain by altering its strategy. It is the stable point of the game when both players are rational and making their best response. Let $NE^S : \Theta^S \rightarrow \mathbb{R}$ denote player 1 's utility when both players play the Nash equilibrium strategy in the SG model. Since the game is a zero-sum game, the utility of player 2 is equal to $-NE^S$.

We compute the Nash equilibrium inspired by the Shapley method [103], which is a dynamic programming approach for finding players' best responses. For game state $\theta^s = \langle t, r, b_1, b_2 \rangle$, the utility of player 1 is equal to sum of the reward that player 1 gets in the current round and the expected utility that he gets in the future rounds. In the future rounds, players will continue to play with their best strategies, so the utility in the future rounds is equal to the one that corresponds to the Nash equilibrium in the future game states. Therefore, the utility of the Nash equilibrium of a game state is as following:

$$\begin{aligned} NE^S(\theta^S) = \max_{a_1^S \in \mathcal{A}^S} \min_{a_2^S \in \mathcal{A}^S} \Big\{ & R^S(\theta^S, a_1^S, a_2^S) + \\ & \sum_{\theta \in \Theta^S} \Phi^S(\theta^S, a_1^S, a_2^S)[\theta] \cdot NE^S(\theta) \Big\} \end{aligned} \tag{5.1}$$

In theory, the game could go for infinite rounds when neither players discloses a vulnerability. In this case, the corresponding utility will be equal to 0, positive infinity or negative infinity. However, for implementation, we need to set a boundary to guarantee that the recursive calculation of Nash equilibrium will stop. We introduce $MAX_t$ to denote the maximum round of the game, and we assume that

$$NE^S(\langle t, r, b_1, b_2 \rangle) = 0, \text{if } t \geq MAX_t. \tag{5.2}$$

### 5.3.3 Optimize the Game After Disclosure

Equation 5.1 is only applicable for calculating the Nash equilibrium of the SG model. Nonetheless, we find an optimized way to compute the Nash equilibrium after the vulnerability is disclosed ($r \neq \emptyset$). The optimized approach is based on the finding that if a player discloses a vulnerability, the other player should attack right after he generates the exploits. We call the player who discloses the vulnerability the explorer, and the other player who witness the disclosure of the vulnerability the observer. Intuitively, disclosure implies that the explorer has discovered the

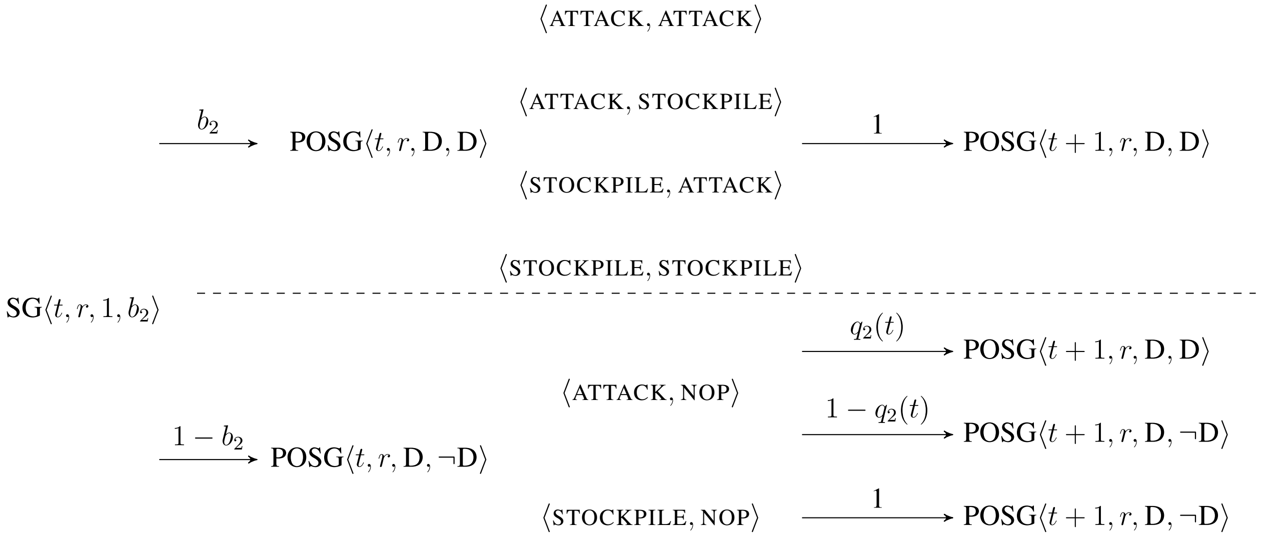


Figure 5.4: The relationship between the SG and the POSG models after disclose a vulnerability. SG⟨·⟩ denotes the game state of the SG model and POSG⟨·⟩ denotes the game state of the POSG model. Suppose player 1 discloses a vulnerability.

vulnerability, and the observer's attack will not reveal to the explorer any new information about the vulnerability. Therefore, there is no collateral damage if the observer attacks, and the observer's best strategy is to constantly attack until his adversary completes patching. We formally prove the finding as follows.

**Theorem 1.** *If one player discloses a vulnerability, the best response of the other player is* $\{\text{D} : \text{ATTACK}, \neg\text{D} : \text{NOP}\}$.

*Proof.* Without loss of generality, we assume that player 1 discloses a vulnerability, and the current game state for the SG model is $SG\langle t, r, 1, b_2\rangle$. The corresponding game state for the POSG model is either $POSG\langle t, r, \text{D}, \text{D}\rangle$ or $POSG\langle t, r, \text{D}, \neg\text{D}\rangle$, shown in Figure 5.4.

If player 2 has not discovered the vulnerability, then the actual game state is $POSG\langle t, r, \text{D}, \neg\text{D}\rangle$. Player 2 can only play NOP, so their action is NOP when they are in state ¬D.

If player 2 has discovered the vulnerability, then the actual game state is $POSG\langle t, r, \text{D}, \text{D}\rangle$. Player 2 chooses actions between ATTACK and STOCKPILE, and the game will deterministically transition to state $POSG\langle t+1, r, \text{D}, \text{D}\rangle$. Recall that $R^P(a_1, a_2)$ represents the utility of player 1 when player 1 chooses action $a_1$ and player 2 chooses action $a_2$. Let $NE^P(\theta^P)$ denote the utility of player 1 in state $\theta^P$ when both players play the Nash equilibrium strategy. Thus, we have

$$\begin{aligned} & NE^P(\langle t, r, \mathrm{D}, \mathrm{D}\rangle) \\ = & \max_{a_1^P \in \mathcal{A}^P} \min_{a_2^P \in \mathcal{A}^P} \left\{ R^P(a_1^P, a_2^P) + NE^P(\langle t+1, r, \mathrm{D}, \mathrm{D}\rangle) \right\} \\ = & \max_{a_1^P \in \mathcal{A}^P} \min_{a_2^P \in \mathcal{A}^P} R^P(a_1^P, a_2^P) + NE^P(\langle t+1, r, \mathrm{D}, \mathrm{D}\rangle) \end{aligned} \tag{5.3}$$

According to the reward matrix in Table 5.3, ATTACK dominates STOCKPILE, and the best strategy for player 2 in state D is ATTACK. Overall, the best strategy for player 2 is $\{\mathrm{D} : \text{ATTACK}, \neg\mathrm{D} : \text{NOP}\}$. □

A special case is that both players disclose a vulnerability at the same round. Under this situation, both players will attack right after disclosure, since both players are the observers and the explorers. As observers, the players will attack once they can; as explorers, the players know how to exploit.

Given the above theorem, the SG model after disclosure becomes a Markov decision process in which the explorer makes a decision given a state of the stochastic game.

Next, we discuss how to compute the best response for the explorer. Given a game state, the explorer chooses one action between ATTACK and STOCKPILE. Suppose player 1 is the explorer.

First, we discuss the algorithm to compute the best response for the POSG game state. If player 2 is in state D, then the game state is $\langle t, r, \mathrm{D}, \mathrm{D}\rangle$. According to Table 5.3, player 1 should play ATTACK:

$$\begin{aligned} NE^P\big(\langle t, r, \mathrm{D}, \mathrm{D}\rangle\big) = {} & R^P(\text{ATTACK}, \text{ATTACK}) + \\ & NE^P\big(\langle t+1, r, \mathrm{D}, \mathrm{D}\rangle\big) \end{aligned} \tag{5.4}$$

Let $\Phi^S(\theta^X)[\theta^Y]$ be the probability that a game transitions from $\theta^X$ to $\theta^Y$. If player 2 is in state $\neg$D and the game is in state $\langle t, r, \mathrm{D}, \neg\mathrm{D}\rangle$, player 1 should choose the action with greater utility according to the following formula:

$$\begin{aligned} & NE^P\big(\langle t, r, \mathrm{D}, \neg\mathrm{D}\rangle\big) = \max_{a_1^P \in \{\text{ATTACK}, \text{STOCKPILE}\}} \Big\{ \\ & R^P(a_1^P, \text{NOP}) + \sum_{\theta \in \Theta^P} \Phi^P\big(\langle t, r, \mathrm{D}, \neg\mathrm{D}\rangle\big)[\theta] NE^P(\theta) \Big\} \end{aligned} \tag{5.5}$$

Finally, given a game state of the SG model $SG\langle t, r, 1, b_2\rangle$, the best response for player 1 is the action with greater expected value of the utilities over POSG states.

$$
\begin{aligned}
NE^S\big(\langle t, r, 1, b_2\rangle\big) = \max_{a_1^P \in \{\text{ATTACK},\text{STOCKPILE}\}} \Big\{ & \\
& b_2 \cdot NE^P\big(\langle t, r, \mathrm{D}, \mathrm{D}\rangle\big) + \\
& (1 - b_2) \cdot NE^P\big(\langle t, r, \mathrm{D}, \neg\mathrm{D}\rangle\big)\Big\}
\end{aligned}
\tag{5.6}
$$

## 5.4 Implementation

In the previous section, we proposed algorithms to calculate the Nash equilibrium of the game. The game is divided into two stages, each of which is solved by dynamic programming. We implemented the code in Python. In this section, we show our pseudo-code in order to convey a clearer structure of the method.

Algorithm 3 shows the calculation for the Nash equilibrium before a vulnerability is disclosed. Given a round index and beliefs of the players, the goal is to compute player utility when both players rationally play their best response. If the round index is equal to or larger than $MAX_t$, which is the maximum number of rounds argument self-configured for the game, then the calculation will stop. Otherwise, the algorithm finds the Nash equilibrium according to Equation 5.1 and Figure 5.3a. For each Nash equilibrium candidate, if players do not disclose the vulnerability, the game will continue in the before-disclosure phase, else the game will step to the after-disclosure phase.

Algorithm 4 shows the computation for the Nash equilibrium after a vulnerability is disclosed. If the game has equal to or more than $MAX_t$ rounds, then the game is over. Otherwise, we update players' state according their APEG skill. If both players have generated the exploit, then both of them should attack. If not, the player who did not disclose the vulnerability should attack once he has generated the exploit. The other player who disclosed the vulnerability should choose between attack and stockpile depending on the sum of the utilities at the current round and that in the future.

```
Input :
 t: The index of the current round
 $b_1, b_2$: The probability that player 1 and player 2 have discovered the vulnerability.
Output:
 $NE^S(\langle t, \emptyset, b_1, b_2 \rangle)$[i]: The utility of player i under the Nash equilibrium at round t before disclosure.
if $t >= MAX_t$ then
    Game is over.
end
$\theta^S \leftarrow \langle t, \emptyset, b_1, b_2 \rangle$;
$\Theta^T \leftarrow$ set of possible states transiting from $\theta^S$;
max $\leftarrow -\infty$;
foreach $a_1^S \in A^S$ do
    min $\leftarrow \infty$;
    foreach $a_2^S \in A^S$ do
        $t \leftarrow R(\theta^S, a_1^S, a_2^S) + \sum_{\theta \in \Theta^S} \Phi^S(\theta^S, a_1^S, a_2^S)[\theta] NE^S(\theta)$;
        if $min > t$ then
            min $\leftarrow$ t;
        end
    end
    if $max < min$ then
        max $\leftarrow$ min;
    end
end
$NE^S(\langle t, \emptyset, b_1, b_2 \rangle)$[1] $\leftarrow$ max;
$NE^S(\langle t, \emptyset, b_1, b_2 \rangle)$[2] $\leftarrow$ -max;
return $NE^S$($\langle$t, $b_1$, $b_2\rangle$);
```

**Algorithm 3:** The before-disclosure game algorithm.

**Input :**

t: The index of the current round

r: The index of the round at which a vulnerability is disclosed

$b_1, b_2$: The probability that player 1 and player 2 have discovered the vulnerability.

**Output:**

$NE^S(\langle t, r, b_1, b_2 \rangle)$: The player utility under the Nash equilibrium at round t after the vulnerability is disclosed at round r.

```
if $t >= MAX_t$ then
    Game is over.
end
foreach $i \in \{1, 2\}$ do
    if $b_i < 1$ and $t > r + \delta_i$ then
        $b_i \leftarrow 1$;
    end
end
if $b_1 == 1$ && $b_2 == 1$ then
    $NE^S(\langle t, r, b_1, b_2 \rangle) \leftarrow NE^P(\langle t, r, D, D \rangle)$;
end
else if $b_2 < 1$ then
    $NE^S\big(\langle t, r, b_1, b_2 \rangle\big) \leftarrow \max_{a_1^P \in \{\text{ATTACK,STOCKPILE}\}} \big\{b_2 \cdot NE^P\big(\langle t, r, \mathrm{D}, \mathrm{D} \rangle\big) + (1 - b_2) \cdot NE^P\big(\langle t, r, \mathrm{D}, \neg\mathrm{D} \rangle\big)\big\}$;
end
else
    $NE^S\big(\langle t, r, b_1, b_2 \rangle\big) \leftarrow \min_{a_2^P \in \{\text{ATTACK,STOCKPILE}\}} \big\{b_1 \cdot NE^P\big(\langle t, r, \mathrm{D}, \mathrm{D} \rangle\big) + (1 - b_1) \cdot NE^P\big(\langle t, r, \neg\mathrm{D}, \mathrm{D} \rangle\big)\big\}$;
end
return $NE^S(\langle t, r, b_1, b_2 \rangle)$;
```

**Algorithm 4:** The after-disclosure game algorithm.

## 5.5 Evaluation and Case Studies

In this section, we apply our algorithm to calculate the Nash equilibrium of cyber-warfare games and discuss the following questions:

- **The Attack-or-Disclose Question (§ 5.5.1).** Previous models [21, 44, 81, 97] limit that a player is allowed to choose only one action which is either attack or disclose. We extend to allow players playing a sequence of actions. Will a player get more utility if he is allowed to play a sequence of actions?
- **The One-Must-Attack Question (§ 5.5.2).** The cyber-hawk model [81] concludes that at least one player will attack. Does our model support this conclusion? If not, is there any counter-example? What causes the counter-example?
- **The Cyber Grand Challenge Study (§ 6.3.1).** How to apply the model to published cyber-conflict events such as the Cyber Grand Challenge, which is a well-designed competition approximating a real-world scenario? Does our model improve the competitor's score if the other players do not change their actions in the game?
- **The $MAX_t$ Effect Evaluation (§ 5.5.4).** How does the configuration of $MAX_t$ affect the results?
- **Performance Evaluation (§ 5.5.5).** What is the runtime performance of the automatic strategic decision-making tool?

We investigate the questions by performing several case studies. Although these cases have concrete parameter values, they characterize general situations in cyber warfare where players have different levels in one or more technical skills.

### 5.5.1 The Attack-or-Disclose Question

Previous models assert that a player should either always attack, or always disclose. However, using our tool, we find cases where a player has a better strategy than to attack or to disclose all the time. For example, consider a game with the parameters in Table 5.4. In this case, player 1's optimal strategy is to disclose and then attack 2 rounds after disclosure.

Intuitively, there are three reasons for player 1 to choose the disclose-then-attack strategy. First, player 1 has more vulnerable resources than player 2, so he will lose if both players attack before disclosure. Second, player 2 has a relatively high ricochet probability, so he will be very likely to generate ricochet attacks if player 1 attacks. Finally, player 1 patches faster than player 2, so he will finish patching earlier, when player 2 is still partially vulnerable. Therefore, player 1 prefers disclose-then-attack strategy over only attacking or only disclosing.

| | Player 1 | Player 2 |
|---|---|---|
| $p_i(t)$ | $\forall t, p_1(t) = 0.5$ | $\forall t, p_2(t) = 0.5$ |
| $u_i(t)$ | $\forall t, u_1(t) = 1$ | $\forall t, u_2(t) = 20$ |
| $q_i(t)$ | $\forall t, q_1(t) = 0.2$ | $\forall t, q_2(t) = 0.9$ |
| $\delta i$ | $\delta_1 = 20$ | $\delta_2 = 20$ |
| $h_i(t)$ | $h_1(t) = 1 - 0.9^t, t < 2$<br>$h_1(t) = 1, t \geq 10$ | $h_2(t) = 1 - 0.1^t, t < 10$<br>$h_2(t) = 1, t \geq 10$ |

Table 5.4: Case I. Player 1's best strategy is to disclose then attack.

### 5.5.2 The One-Must-Attack Question

Previous work concludes that at least one player must attack [81]. However, we argue that the conclusion is inaccurate, by showing cases in which *neither* player prefers attacking. Consider the game with the settings shown in Table 5.5. We find that both players will choose to PATCH after they find the vulnerability. The intuition is that player 1 should never choose to ATTACK because he will suffer a greater loss if player 2 launches ricochet attacks. Player 1 should also never choose to STOCKPILE, because player 2 may re-discover the vulnerability and then ATTACK. Therefore, player 1's best strategy is to PATCH once he discovers the vulnerability. After player 1 discloses a vulnerability, player 2 receives the patch and generates exploits based on the patch, which costs him $\delta_2$ rounds. Within the rounds, player 1 would have completely patched his own machines, which makes any future attack from player 2 valueless.

Furthermore, we observe two necessary elements leading to players' not attacking strategy: ricochet capability and patching capability. To illustrate our observation, we computed the Nash equilibrium of two other games, where we only changed the value of the ricochet or patching parameters, and we found that one player will prefer attacking in new games.

First, we consider the scenario excluding ricochet. We keep the parameters Table 5.5, but set $q_i(t) = 0, \forall t$. We observe that both players will attack until the end of the game. Because ATTACK always bring positive benefit while STOCKPILEand NOP always bring 0 benefit, ATTACK dominates STOCKPILEand NOP at any round. Therefore, the optimal strategy for both players is to attack as soon as they discover the vulnerability.

Second, we consider the scenario where one player slows down his patching speed. Suppose we replace the original patching function $h_2(t)$ with $h_2(t) = 1 - 0.1^t, t < 20$ and $h_2(t) = 1, t \geq 20$. We observe that the best strategy for player 1 is to attack, since some of the player 2's resources remain vulnerable after player 1 is done with patching. This case indicates that even though the ricochet attack exists, if a player does not patch fast enough, he will still be attacked by his opponent. In conclusion, we find that both ricochet and speedy patching are necessary in order to prevent adversaries from attacking.

| | Player 1 | Player 2 |
|---|---|---|
| $p_i(t)$ | $\forall t, p_1(t) = 0.8$ | $\forall t, p_2(t) = 0.01$ |
| $u_i(t)$ | $\forall t, u_1(t) = 2$ | $\forall t, u_2(t) = 20$ |
| $q_i(t)$ | $\forall t, q_1(t) = 0.2$ | $\forall t, q_2(t) = 0.9$ |
| $\delta i$ | $\delta_1 = 20$ | $\delta_2 = 20$ |
| $h_i(t)$ | $h_1(t) = 1 - 0.9^t, t < 10$<br>$h_1(t) = 1, t \geq 10$ | $h_2(t) = 1 - 0.1^t, t < 10$<br>$h_2(t) = 1, t \geq 10$ |

Table 5.5: Case II. Both players' best strategy is to disclose without attacking.

### 5.5.3 The Cyber Grand Challenge Study

The Cyber Grand Challenge (CGC) is an automated cyber-security competition designed to mirror "real-world challenges" [47]. This competition provides an excellent opportunity to evaluate the strategies suggested by our model against those actually carried out by competitors. The CGC final consists of 95 rounds. In this case study, we experimented on the ranking of the third-place team in the Cyber Grand Challenge, Shellphish. Based on their public discussions regarding their strategy, Shellphish simply attacked and patched right away [48]. This made them an optimal subject of this case study, as, since they would use their exploits as soon as possible (rather than stockpiling them), we can closely estimate their technical acumen for the purposes of testing our model. We call our modified, more strategic, player "Strategic-Shellphish".

In our experiment, we adapted our model to the CGC final in the following way. First, we update the reward function on the CGC scoring mechanism. As the CGC final is not a zero-sum game, we compute the Nash equilibrium by focusing on the current round. Second, we separate the game by binaries, and for each binary we model Strategic-Shellphish as one player while all the non-Shellphish team as the other player. Third, we estimated the game parameters according to the data from the earlier rounds, then calculated the optimal strategy and applied the strategy in the later rounds. For example, we get the availability score for the patch by deploying it in the earlier rounds. The data is from the public release from DARPA, which includes the player scores for each vulnerable binaries in each round.

In the first CGC experiment, we estimated the game parameters by the information of the first 80 rounds of the game, and applied the model on the 80-95 rounds. This range included 11 challenge binaries, and we simulated Shellphish's performance, if they had used our model for strategy determinations, across these programs. The score comparison for each vulnerability is shown in Figure 5.5, with the x axis representing the 11 binaries and the y axis representing the scores. The new scores are either higher or equal to the original score. Among these binaries, our model helps improve 5 cases out of 11. The overall score for the 11 vulnerabilities is shown in Figure 5.6. The original Shellphish team got 38598.7 points, while our model got 40733.3 points. Moreover, our Strategic-Shellphish team won against all other teams in terms of these 11 vulnerabilities.

We observed that Strategic-Shellphish withdrew the patch of some vulnerabilities after the

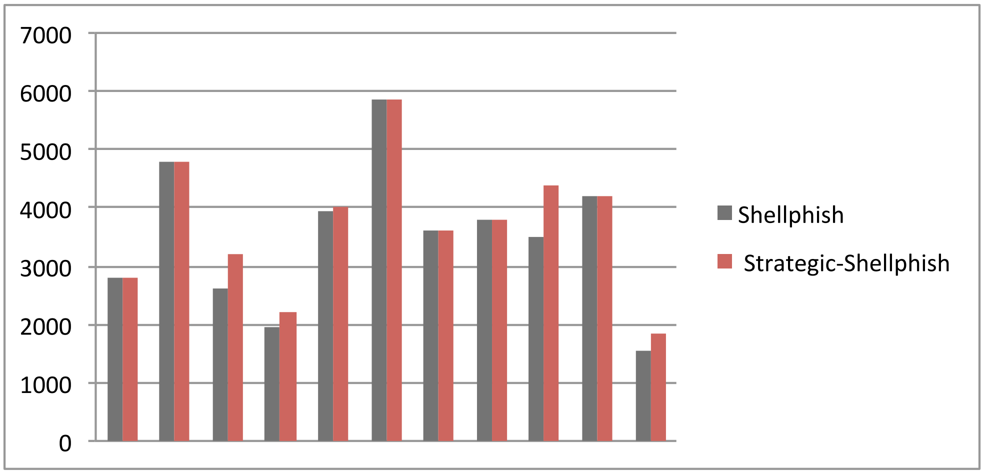


Figure 5.5: The per-vulnerability score comparison between the original Shellphish team and Strategic-Shellphish – the Shellphish team + our model.

first round of patching. After the first round of patching, Strategic-Shellphish got the precise score for availability, and this helped it compare the cost of patching to the expected lost in the future rounds.

In the second CGC experiment, we estimated the game parameters by the information of the first 15 rounds. Given the parameters, Strategic-Shellphish calculates the likelihood that the other teams discovers the vulnerability, and it uses our algorithm to determine the best response. Before it is well-aware of the patching cost, we assigned the cost to 0. After the first round of patching, we updated the patching cost and adjusted the future strategy.

The score for the entire game is shown in Figure 5.7. The original Shellphish team got 254,452 points and ranked third in the game. On the other hand, the Strategic-Shellphish got 268,543 points, which is 6000 points higher than the score of the original 2nd-rank team. Our experiment highlights the importance of our model as well as the optional strategy solution. If a team such like Shellphish used our model, it could have achieved a better result compared to its original strategy. In fact, in the Cyber Grand Challenge, the difference between third (Shellphish) and second (Strategic-Shellphish) place was $250,000.

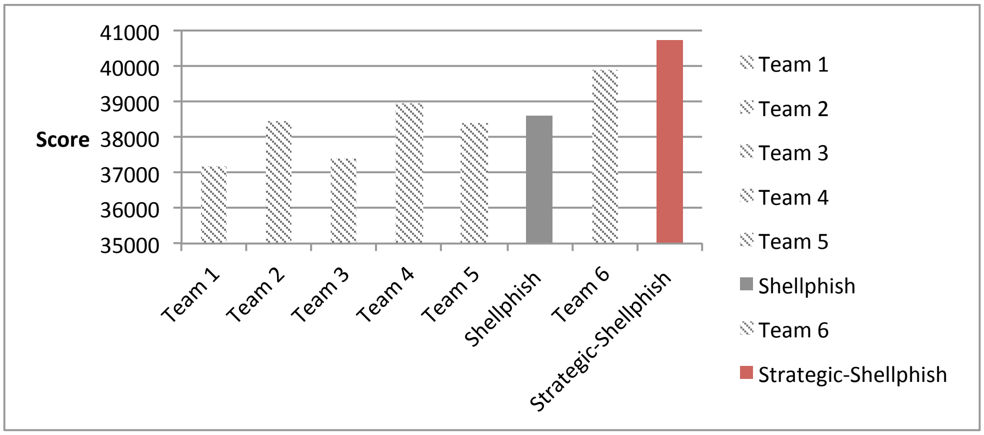


Figure 5.6: The overall score comparison between the original Shellphish team and Strategic-Shellphish for the game round 80-95.

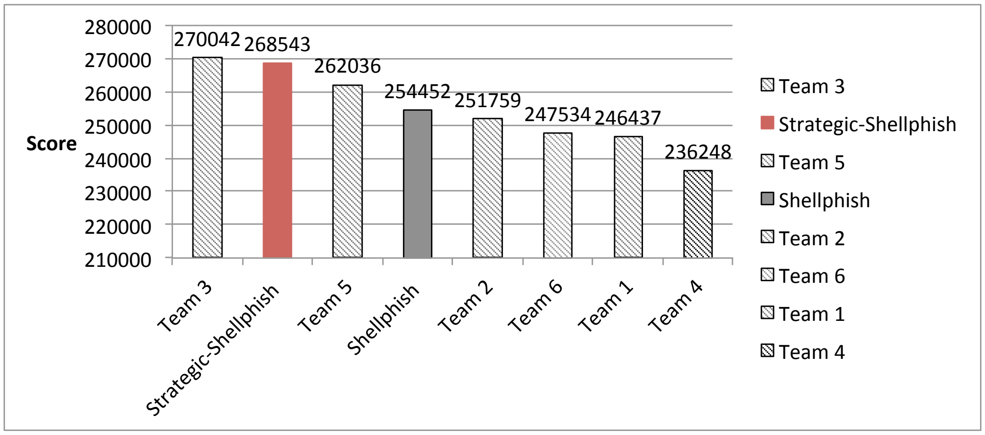


Figure 5.7: The overall score comparison between the original Shellphish team and Strategic-Shellphish for the entire game.

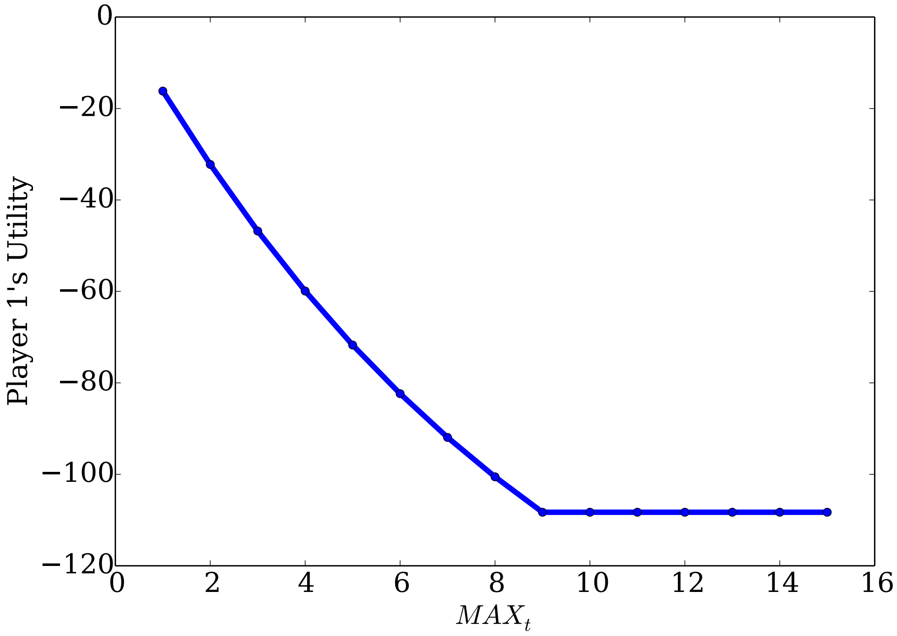


Figure 5.8: Player 1's Utility over different maximum number of round ($MAX_t$).

### 5.5.4 The $MAX_t$ Effect Evaluation

To understand the effect of $MAX_t$ on the final result, we fixed the parameter values in Table 5.4 and varied the value of $MAX_t$ from 1 to 15. For each game, we computed the Nash equilibrium and its corresponding players' utilities. As the game is a zero-sum game and the utility of player 2 is always symmetric to that of player 1, we will focus on player 1's utility.

Figure 5.8 shows player 1's utility. We observed that when $MAX_t$ is small, the change of $MAX_t$ will affect the Nash equilibrium and players' utilities. As $MAX_t$ becomes larger, the change of $MAX_t$ will no longer affect the Nash equilibrium, and the players' utilities will become stable. In our case, when $MAX_t = 1$, player 1 will patch and player 2 will attack for both rounds. When $MAX_t = 2$, player 1 will disclose at the first round and attack at the second round, while player 2 will attack for both rounds, and, meanwhile, patch if player 1 discloses the vulnerability. When $MAX_t \geq 3$, player 1 will disclose at the first round and attack since the third round. This observation implies that players tend to be more aggressive in a shorter game. It also explains why the result of the cyber-hawk model [81] is suboptimal: if the game

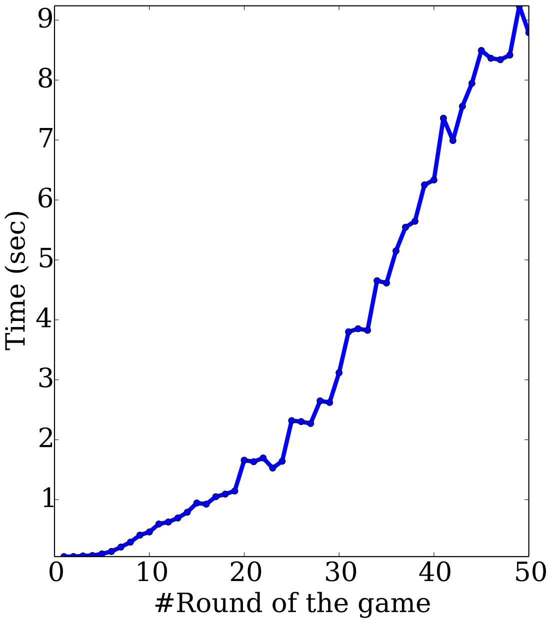


Figure 5.9: The time (in second) for computing the Nash equilibrium over different maximum number of round $MAX_t$.

is considered as a single-round game, players will neglect the loss in the future and make a local optimal strategy rather than a global optimal one.

### 5.5.5 Performance Evaluation

In this evaluation, we fixed the player parameters and measured the time for computing the Nash equilibrium under different $MAX_t$ values from 1 to 50. We show the time performance in Figure 5.9. Based on the figure, we found that although the time of computing increases as $MAX_t$ grows, our tool is able to find the Nash equilibrium of all games within seconds. In practical, $MAX_t$ needs to be configured properly in order to balance between the action frequency (i.e., should a player act per minutes or per day?) and the action performance (i.e., how long do we need to respond to the zero-day vulnerability events?).

## 5.6 Discussion and Future Work

Our work advances zero-day strategy research by constructing a game model covering features that are of great significance in real-world cyber warfare, such as activity over multiple rounds, partial availability of information, and emergent offensive techniques like APEG and ricochet. In this section, we discuss some aspects to be addressed in future work.

**Irrational Players and Collusion.** We focus on a setting with rational players engaged in zero-sum games. It is well established that governments and people act irrational from time to time. Nonetheless, an analysis of rational behavior highlights an important consideration point. We leave the modeling of non-rational behavior and non-zero-sum games as future work.

**Parameter Sensitivity.** Our model employs parameters to capture players' different skill levels. These parameters need to be evaluated, and one way is to use a relevant benchmark proposed by Axelrod et al. [21]. For example, one can estimate $p_i(t)$ by reasoning about the ratio of the vulnerabilities independently rediscovered in software from the research by Bilge et al. [27]. As the White House stated that the government weighed the likelihood that the other nations rediscover the vulnerability, the evaluation approach should have already existed. In the future, we need to investigate the robustness of the model under different parameter margins.

**Multiple Nash Equilibria.** It is possible that multiple Nash equilibria exist in a cyber-warfare game. However, due to the zero-sum game property, player utilities remain the same for all Nash equilibria. Therefore, any Nash equilibrium is players' optimal strategy. Although we do not discuss the entire set of possible Nash equilibria, finding them all is a straightforward extension of our algorithm, in the way that one could record all strategies with same value as the maximum one.

**Deception.** Although the game parameters are public, they can be manipulated by players. For example, a player could pretend to be weak in generating exploits by never launching any attack against anyone. In our dissertation, we do not consider player deception, and we leave it for future work.

**Inferring Game State from Parameters.** We consider that players infer the states of other players by detecting attacks or learning vulnerability disclosure. However, we do not consider that players could these state by reasoning about game parameters. For example, suppose we have a game with public game parameters denoting that player 1 is able to capture all the attacks and player 2 should always attack after he generates an exploit. In this case, if player 1 does not detect attacks from player 2, then player 2 has not generated an exploit, and the belief on player 2 should be zero all the time until player 1 detects an attack. To tackle this problem, a possible way is to solve the game separately with different groups of parameter conditions.

**Multiple Vulnerabilities.** Our game model focuses on one vulnerability. We assume that vulnerabilities are independent, and a game with multiple vulnerabilities can be viewed as separate games each of which has a single vulnerability. To combine multiple vulnerabilities in a game, a possible direction is to consider modification of the game parameters. For example, instead of the probability that the opponent re-discovers the vulnerability, we could use the probability that the opponent discovers *any* vulnerabilities. Also, we could extend the utility function by including the utility gained from other vulnerabilities.

**Limited Resources.** When players are constrained by limited resources, they may have fewer strategy choices. For example, if a player has limited resources, he may not be able to simultaneously generate an exploit and generate a patch. The limited resources may affect players' best response as well as the Nash equilibrium. In the future, we need to come up with updated model and algorithm to address this issue.

**The Incentives of Patching.** In our model, we consider patching as a defensive mechanism that only prevents players from losing utility. This leads to players not having incentives to disclose a vulnerability. We argue that patching might bring positive benefits to players. For instance, a player would have a better reputation if he chooses to disclose a vulnerability and patch their machines. We leave the consideration of the positive reputation caused by disclosure as future work.

## 5.7 Conclusion

In this chapter, we present a cyber-warfare model which considers strategies over time, addresses players uncertainty about their opponents, and accounts for new offensive and defensive techniques that can be employed for cyber-warfare, e.g., the ricochet attack and APEG. We propose algorithms for computing the Nash equilibrium of the model, and our algorithm is able to find better strategies than previous work within seconds. Moreover, by solving the game model, we allow decision makers to calculate utility in scenarios like patch-then-exploit, as well as show where, in the parameter space of the game model, it makes more sense to patch than to attack. Our model also challenges previous results, which conclude that at least one player should attack, by showing scenarios where attacking is not optimal for either player.

# Part III

# Investigation

# Chapter 6

# Qualitative Technique Evaluation on Autonomous Computer Security Games

In previous chapters, we have presented the autonomous computer security game model and the algorithm for finding the optimal strategy in the game model. We have also introduced multiple software security related techniques such as function recovery for stripped binaries and exploit reuse and shellcode transplant.

In this part of the dissertation, we investigate how techniques qualitatively change the outcome of software security. More specifically, we consider software security instances that can be modeled as autonomous computer security games, and evaluate how the Nash equilibrium and utility of an autonomous computer security game changes due to a particular technique, given the assumption that players in the game are rational, playing their optimal strategy.

This chapter is structured as follows. First, we introduce the methodology of how to qualitatively evaluate a software security-related technique (Section 6.1). Then we investigate how the parameters in autonomous computer security game influence Nash equilibrium and expected utility (Section 6.2). Finally, we take the Cyber Grand Challenge (CGC) final as a concrete case and investigate how techniques such as ByteWeight and ShellSwap change the game result (Section 6.3.1).

## 6.1 Introduction

Evaluating techniques is essential because evaluation justifies and measures the merit of techniques. Traditionally, techniques are evaluated in a quantitative way. For example, a vulnerability discovery technique is evaluated by how many vulnerabilities are found using the technique, and a vulnerability patching technique is evaluated by how many vulnerabilities are successfully patched against attacks.

On the other hand, qualitative evaluation of how techniques change security outcome is also very important in security study [64]. Security researchers have pointed out that an important step forward in security research is to "stop insisting that quantitative is better than qualitative; both types of measurement are useful" [43, 88, 89].

However, little research has discussed the *methodology* of how to evaluate techniques qualitatively, not mention the result about how software security-related techniques help to achieve security. For example, the Offensive Defense is known as a famous doctrine for software security research. Parties who are more skillful in the offense skills will find more vulnerabilities in an earlier time, and these parties will be more proactive in defense. However, as more offensive techniques are published as papers or even released with source code, those offensive techniques do not exclusively belong to some parties. Instead, those published techniques are known and can be used by all parties. Under such circumstance when everyone increases its attack ability, will the technique still make the world more secure? Does "offensive defense" still hold? In this section, we will introduce the approach to answering such question, and we will investigate multiple binary analysis techniques.

## 6.2 Evaluating a Technique Class on the Autonomous Computer Security Game Model

In this section, we qualitatively evaluate a class of software analysis techniques on autonomous computer security game model. For a class of software analysis techniques, we study whether and how the development of such class impacts the outcome of software security.

Answering this question helps us to understand the benefit of developing a technique class. Moreover, by comparing the benefit of different technique classes, an agent, such as a company or a country, will be able to prioritize the development of techniques. For example, given a fixed budget, a nation can use our method to investigate the benefits of defense and offense technique classes and decide how to allocate resources for the two classes.

A class of techniques includes all techniques serving for the same purpose. For example, Exploit generation is a class of techniques, which includes all techniques that find vulnerabilities and produce exploits such as Driller [110], AEG [19], Mayhem [37], Q [99] and AFL [17].

Software security is abstracted as a model. In this model, a class of techniques is abstracted as one or multiple parameters. For example, in the cyber-hawk model [81], the exploit generation technique class is modeled as one parameter. Meanwhile, the outcome of software security is represented as a function related to the parameters. For example, one security outcome is players' optimal strategy. For a game model such as cyber-warfare [24], Nash Equilibrium is the strategy profile when all players are playing optimally. Therefore, we use Nash Equilibrium to represent the function for the security outcome.

Essentially, the development of a technique in a technique class changes the value of the related parameters, and the change of the parameters leads to the change of the value of the representing function. In consequence, answering whether and how the development of a technique class impacts security outcome, is equivalent to answering how the parameters, which behaves as variables, change the value of the function.

### 6.2.1 Case Study: Exploit Reuse

We evaluate the exploit reuse technique class on software security. In specific, we investigate how exploit reuse changes the disclosure of vulnerability by the following questions:

- Does exploit reuse keep a player from being stealthily attacked by its opponent?
- Does exploit reuse stop *both* players from stealthily attacking each other?

To answer these questions, we use the autonomous computer security game (Chapter 1.3) as the abstract software security model. Recall that the autonomous computer security game model considers five parameters:

- $p_i(t)$: The probability distribution over time that player $i$ discovers a vulnerability at round $t$.
- $q_i(t)$: The probability to launch a ricochet attack with exploits that player $i$ received in the previous round.
- $h_i(t)$: The ratio of the amount of patched vulnerable resources over the total amount of vulnerable resources by $t$ rounds *after* the vulnerability is disclosed.
- $\delta_i$: The number of rounds required by player $i$ to generate a patch-based exploit after a vulnerability is disclosed.
- $u_i(t)$: The dynamic utility that player $i$ gains by attacking his opponents at round $t$.

In the model, parameter $q_i(t)$ represents exploit reuse, as exploit reuse is a kind of ricochet attack. Therefore, we treat $q_i(t)$ as a variable. Vulnerability disclosure is related to players' strategy. Assuming players are playing rationally, then vulnerability disclosure and players' final strategy are related to Nash Equilibrium, and the representing function for security outcome is the Nash Equilibrium of the game model.

For the other parameters, we treat patching function ($h_i(t)$), utility function ($u_i(t)$) and patch-based exploit generation ($\delta_i$) as control variables by setting equal values for both players. We set the vulnerability discovery function as variable function and calculate the Nash Equilibrium under different values of vulnerability discovery function and exploit reuse.

Note that we should treat all the other parameters as variables in theory. However, there are infinite numbers of functions, and the domain of patch-based exploit generation is unlimited, and thus enumerating patching function, utility function and patch-based exploit generation parameters is theoretically impossible. Also, those parameters are not of primary interests since they are independent of exploit reuse. Therefore, in the evaluation we held constant and equal values for the parameters of both players. Setting equal values for both players avoids Nash Equilibrium being influenced by the difference of these elements. However, the value itself may also affect Nash Equilibrium. In this dissertation, we use this case for the purpose of demonstration for qualitatively evaluating a technique class for security outcome. We leave the study for more parameters (which essentially, is the investigation of the impact of higher dimensional variables for the Nash Equilibrium of the autonomous computer security game model) as future work.

In specific, we set utility function as $u_1(t) = u_2(t) = 1, \forall t$ and patch-based exploit generation parameter $\delta_1 = \delta_2 = 0$ throughout our evaluation. We set patching ratio function as $h_1(t) = h_2(t) = 1 - 0.5^t$, due to that a patching ratio function normally limits to 1 (patching should approach to be complete). To scope vulnerability discovery function from infinite enumeration with limited domain, we set $p_1(t) = p_1, \forall t$ and $p_2(t) = p_2, \forall t$, and we call $p_1$ and $p_2$ *vulnerability discovery rates*.

**Does exploit reuse keep a player from being stealthily attacked by its opponent?** To answer this question, we need to figure out the relationship between players' stealthy attack and the representing function, Nash equilibrium. A stealthy attack occurs when a player plays $\{\text{D} : \textsc{attack}, \neg\text{D} : \textsc{nop}\}$before any players play $\{\text{D} : \textsc{patch}, \neg\text{D} : \textsc{nop}\}$. The action is made according to a Nash equilibrium. A Nash equilibrium is a strategy profile denoting players' optimal strategy. For autonomous computer security games, as partial observation stochastic games, a player's strategy is a *sequence*, where each element denotes the actions under two different player states. For example, suppose Player 1's optimal strategy for an autonomous computer security game with 2 rounds is

$\{\mathrm{D} : \textsc{attack}, \neg\mathrm{D} : \textsc{nop}\}, \{\mathrm{D} : \textsc{patch}, \neg\mathrm{D} : \textsc{nop}\}$

This means that Player 1 plays $\{\mathrm{D} : \textsc{attack}, \neg\mathrm{D} : \textsc{nop}\}$before he plays $\{\mathrm{D} : \textsc{patch}, \neg\mathrm{D} : \textsc{nop}\}$, and if Player 2 does not play $\{\mathrm{D} : \textsc{patch}, \neg\mathrm{D} : \textsc{nop}\}$at the first round, then Player 1 stealthily attacks Player 2. Therefore, to check stealthy attack, we need to check if $\{\mathrm{D} : \textsc{attack}, \neg\mathrm{D} : \textsc{nop}\}$exists before the round when one player plays $\{\mathrm{D} : \textsc{patch}, \neg\mathrm{D} : \textsc{nop}\}$, after we calculate the Nash equilibrium.

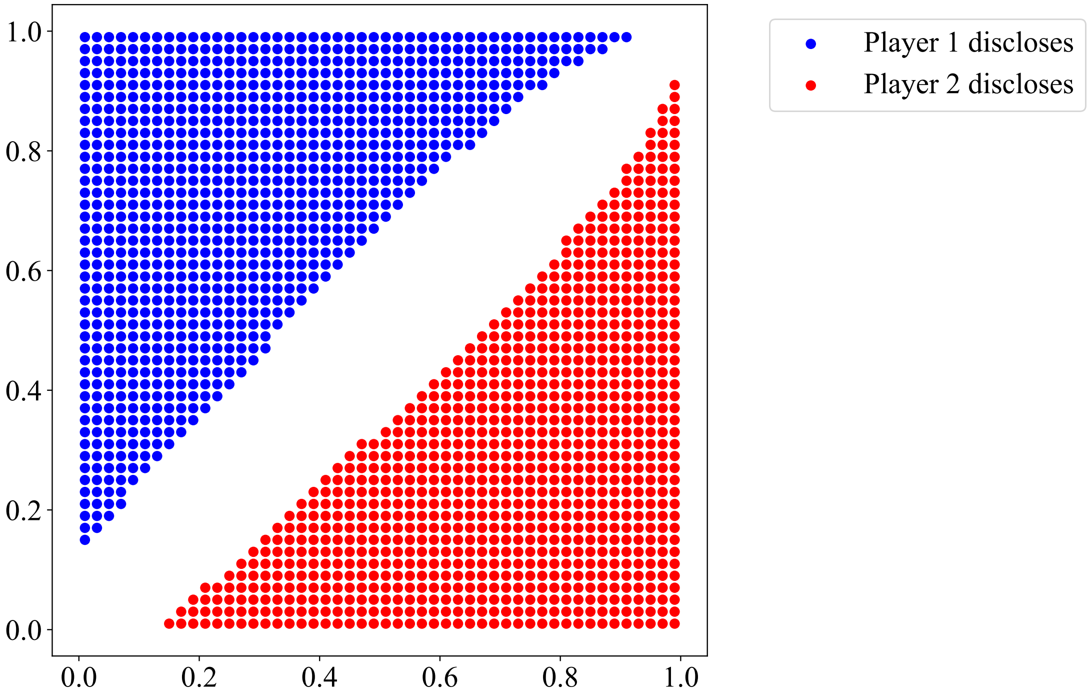


Figure 6.1: Vulnerability disclosure without exploit reuse.

Figure 6.1 shows vulnerability disclosure associating with players' vulnerability discovery rate. In the chart, the x-axis and the y-axis denote Player 1's and Player 2's vulnerability discovery rates, respectively. Each dot represents the case where Player 1's vulnerability discovery rate is equal to its x-axis value and Player 2's vulnerability discovery rate is equal to its y-axis value. Blue dots means Player 1 discloses the vulnerability, and red dots means Player 2 discloses the vulnerability.

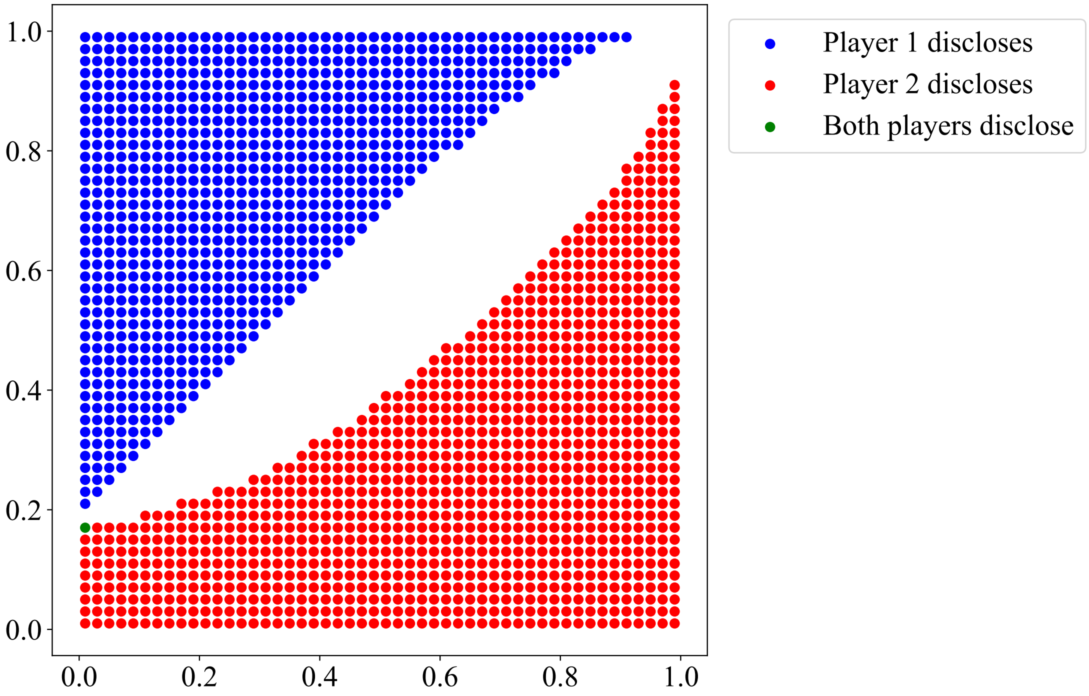


Figure 6.2: Vulnerability disclosure with exploit reuse function $q_1(t) = 0.1, \forall t$ and $q_2(t) = 0, \forall t$.

Based on the figure, we find that when two players have equal or similar vulnerability discovery rate, they do not disclose the vulnerability. When a player's vulnerability discovery rate is higher than the other player to some extent, the player with lower vulnerability discovery rate discloses the vulnerability. Under this setup, at least one player always attacks the other player, which is consistent with the observation of previous work [81].

Interestingly, we observed that a vulnerability is always disclosed at the beginning of the game under all the enumeration. Our understanding is that this is related to the setup of the utility function and patching function. Intuitively, for a player who prefers patching, early patching is always better than late patching, unless attacking later helps him getting more utility from his opponent, meaning that the opponent's utility function increases in later rounds. We leave the formal proof as the future work.

Figure 6.2, Figure 6.3 and Figure 6.4 show the vulnerability disclosure associating with different vulnerability discovery rates. In these figures, we disable Player 2's exploit reuse skill

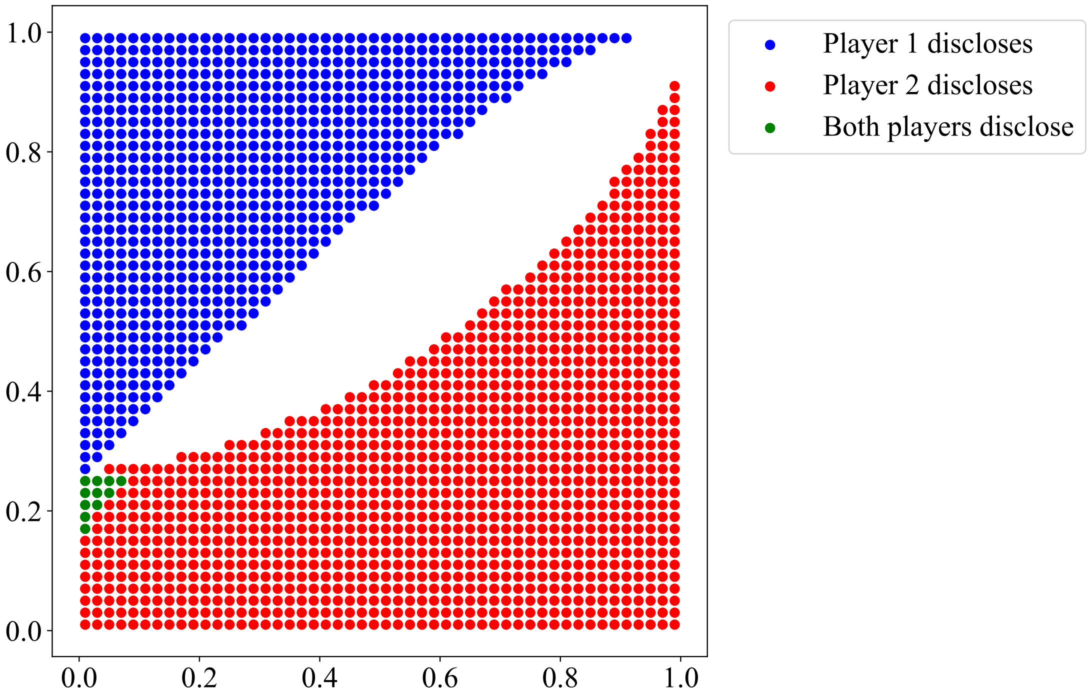


Figure 6.3: Vulnerability disclosure with exploit reuse function $q_1(t) = 0.2, \forall t$ and $q_2(t) = 0, \forall t$.

by setting the function constantly zero, and we change Player 1's exploit reuse function from $q_1(t) = 0.1, \forall t$ to $q_1(t) = 0.3, \forall t$.

Based on the figure, we find that exploit reuse stops a player from attacking, if its opponent is capable of exploit reuse. The intuition is that, if a player's opponent can reuse an exploit, then the player should consider collateral damage if he attacks, and when the collateral damage is greater than the expected attacking utility – which depends on both players' vulnerability discovery skills – the player should disclose the vulnerability instead of stealthily attack.

**Does exploit reuse stop *both* players from stealthily attacking each other?** In the previous discussion, we find that although exploit reuse helps a player from being stealthily attacked by the other player, there is still at least one player stealthily attack. As a followup question, we study whether exploit reuse stops *both* players stealthily from attacking each other.

Figure 6.5, Figure 6.6, Figure 6.7 and Figure 6.8 shows the vulnerability disclosure of the games with different attacking utility when *both* players have the skill of exploit reuse. For ex-

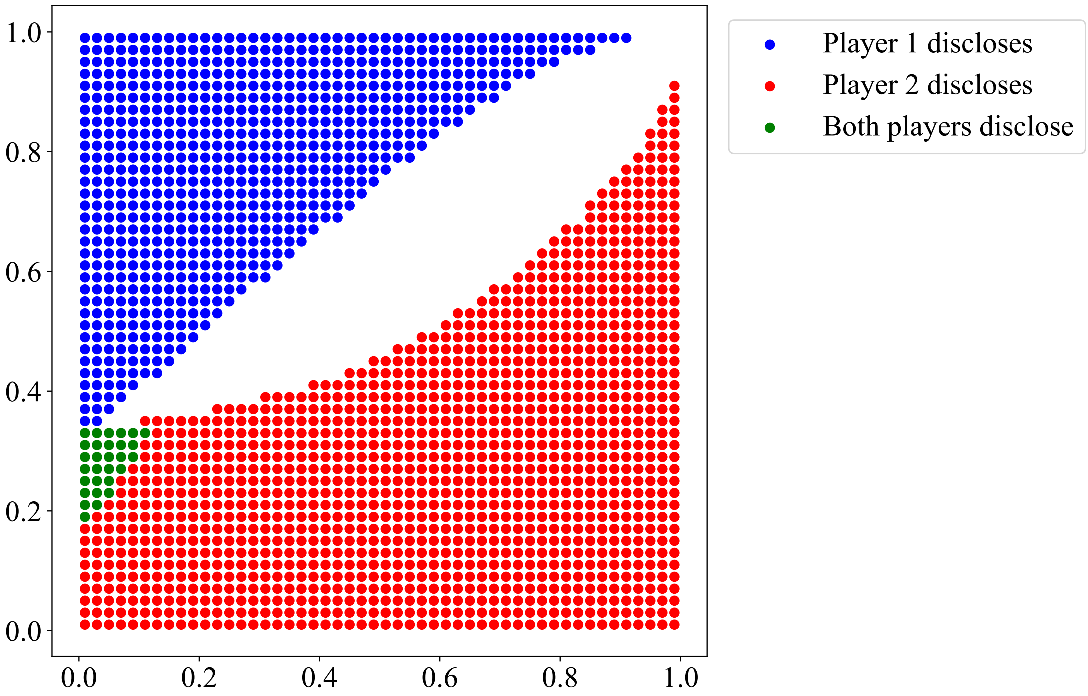


Figure 6.4: Vulnerability disclosure with exploit reuse function $q_1(t) = 0.3, \forall t$ and $q_2(t) = 0, \forall t$.

ample, if an exploit reuse technique is published through an academic conference, then with the publicity all players will obtain the knowledge of the technique. We observed that when both players are capable of exploit reuse, there are circumstances that both choose to disclose the vulnerability. As the exploit reuse skill increases, the both-disclose circumstances also increases. When both players have full exploit reuse skill (meaning that both players are able to reuse all exploits), at least one player discloses the vulnerability. Therefore, we conclude that when *both* players have the exploit reuse skill, exploit reuse stops both players from stealthily attacking. This observation is encouraging; it illustrates the importance of developing and *publishing* exploit reuse techniques. When everyone has the full capacity of exploit reuse, there will be no zero-day attack in the world.

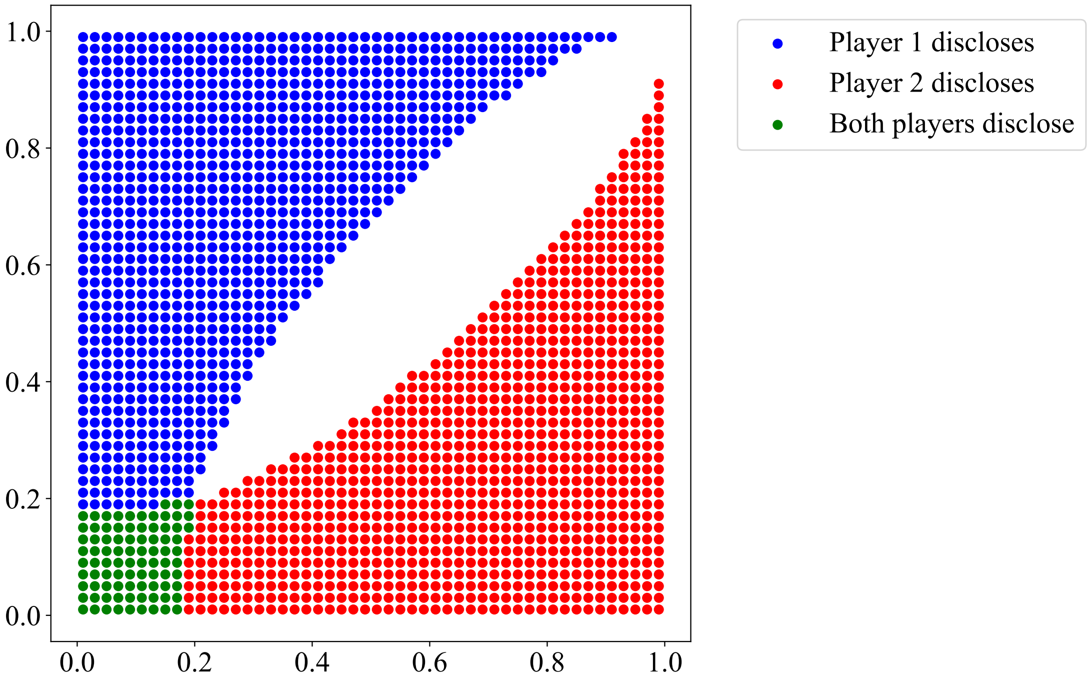


Figure 6.5: Vulnerability disclosure with exploit reuse function $q_1(t) = q_2(t) = 0.1, \forall t$.

## 6.3 Evaluating a Technique on a Security Case

In this section, we discuss qualitative technique evaluation on concrete security cases. The goal of the evaluation is to investigate a technique's security contribution to a specific security scenario. Given a technique and a security case, the evaluation answers whether and how a technique changes the security outcome of the case.

We first introduce the approach to the qualitative technique evaluation. Similar to the technique class evaluation, we need to abstract security cases to a model, identify the model parameters related to the evaluating technique and define the function representing the security outcome. In addition, we calculate the parameter values under two conditions: when the evaluating technique presents and does not present. Then we calculate the value of the representing function associated with the two parameter values. Finally, we compare the value of the representing function and the difference of the representing function shows the impact of the technique on the concrete security case.

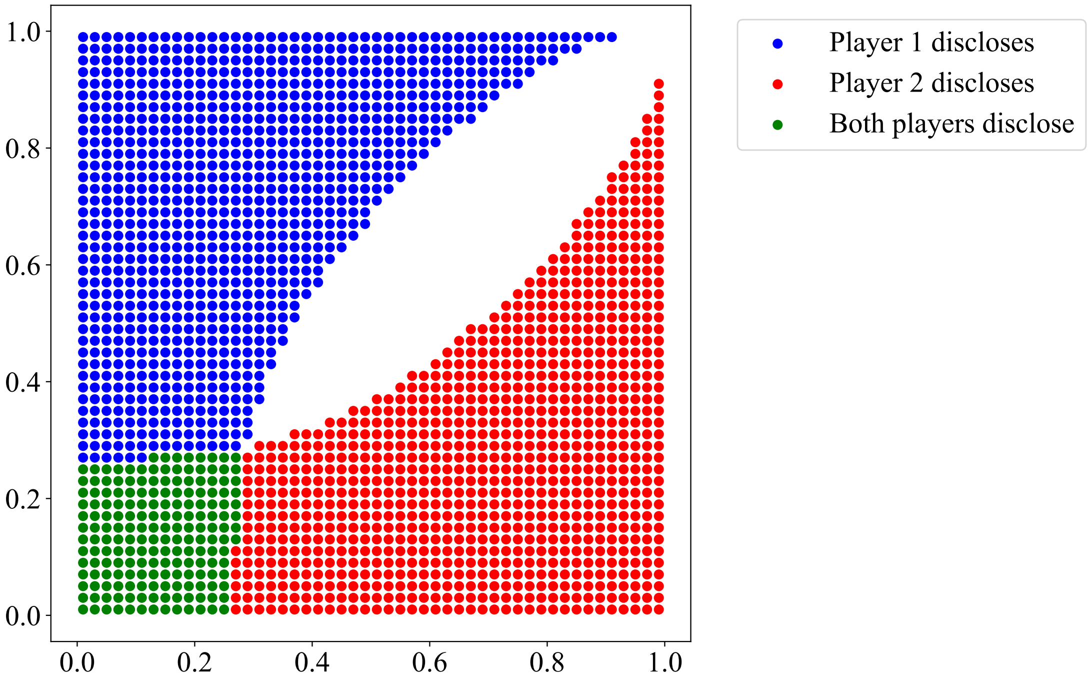


Figure 6.6: Vulnerability disclosure with exploit reuse function $q_1(t) = q_2(t) = 0.2, \forall t$.

### 6.3.1 Case Study: ByteWeight on Cyber Grand Challenge Final

As a case study, we evaluate ByteWeight on Cyber Grand Challenge Final. Cyber Grand Challenge is an offense-defense competition hosted by DARPA in 2016. In the final of the competition, 7 teams played and competed with each other. Each team has an identical server running multiple programs. Those programs contain vulnerabilities, and the teams need to find the vulnerabilities, attack the other teams using the vulnerabilities and defend their own servers from being exploited. The final is composed by 95 rounds. In each round, a team is allowed to attack the other teams for one time. If the attack is successful, then the team wins the offense score. Meanwhile, a team can also patch their own programs. If the patch is tested to be effective, then the team gets defense score. If the patch is also tested to be robust, meaning that the patched program passes performance and functionality test, then the team also gets performance score. The total score is calculated by combining offense score, defense score and performance score.

In Cyber Grand Challenge final, players are represented by autonomous cyber reasoning

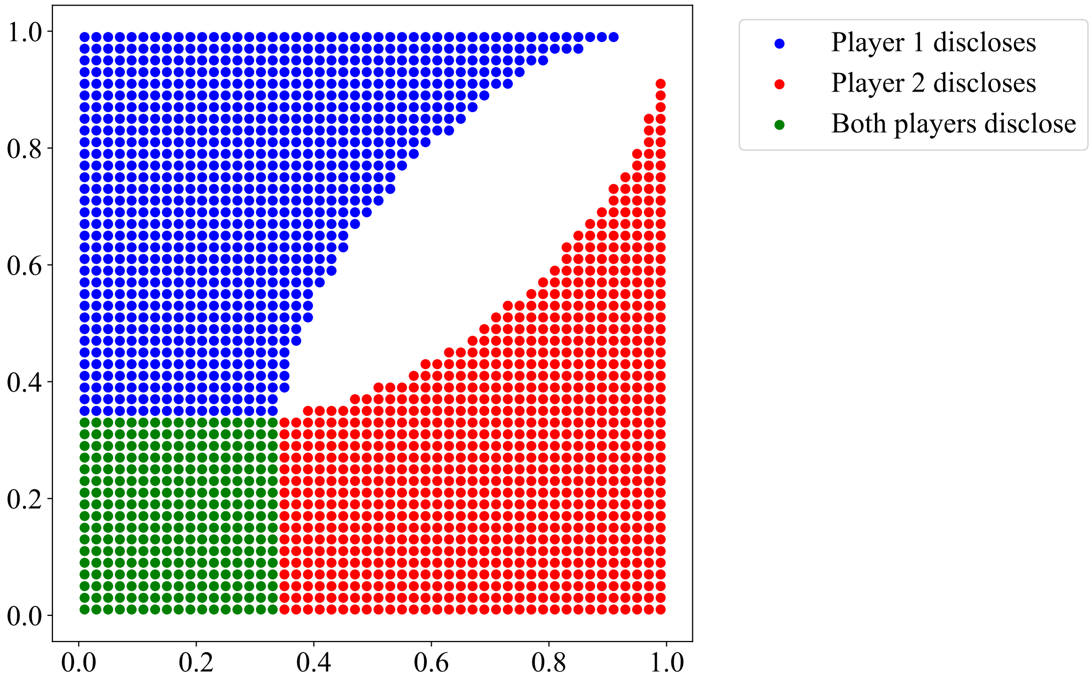


Figure 6.7: Vulnerability disclosure with exploit reuse function $q_1(t) = q_2(t) = 0.3, \forall t$.

systems with no human intervention. Under this setup, the systems discover vulnerabilities and decide what to do with the vulnerability. Due to the complete publishing of the data and some of the autonomous cyber reasoning systems, we use the data from Cyber Grand Challenge final for some of the evaluation of our work. However, note that our research is not limited to the Cyber Grand Challenge game; it is general for real-world scenarios such as nations and companies strategy for zero-day vulnerabilities, as well as programs of daily use such as `coreutils` and `binutils` packages.

In specific, we evaluate how ByteWeight contributes for team Shellphish in the Cyber Grand Challenge final. ByteWeight [23] (Section 3) is a function identification technique. It was adopted to team Shellphish's system, Mechanical Phish, and the technique served for binary patching. The representing function is Shellphish's score. To evaluate ByteWeight, we disabled ByteWeight, re-ran the patching component of Mechanical Phish and found the binaries with failed patching [46]. We then calculated the strategy of these binaries, and we updated Shellphish's score with the new strategy. In the last, we compared Shellphish's score with the one

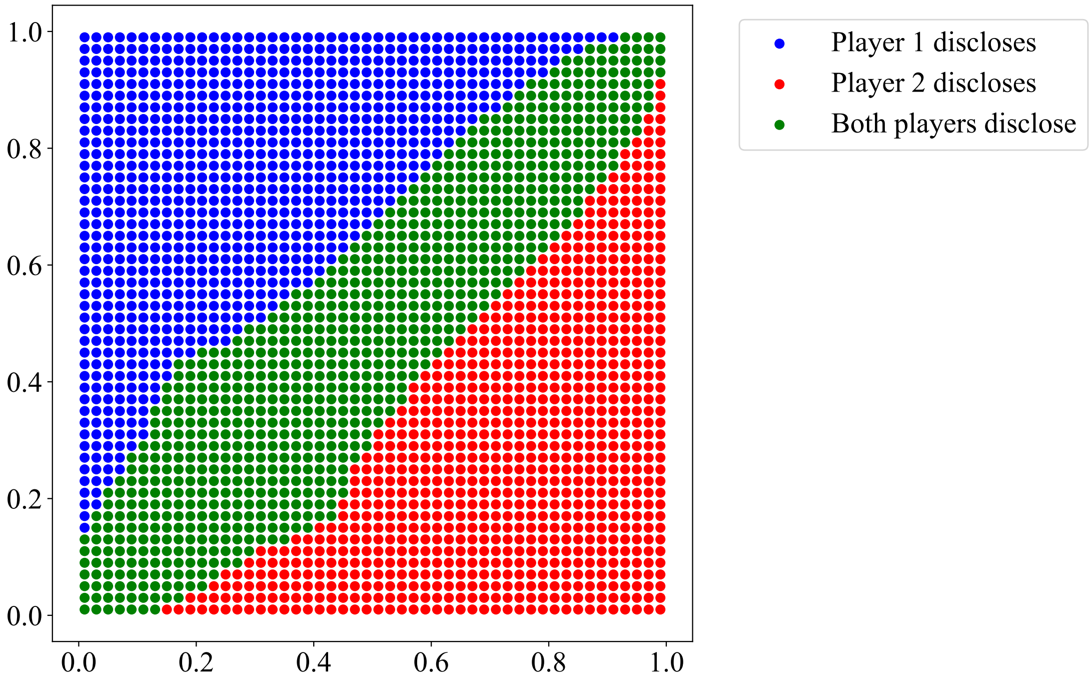


Figure 6.8: Vulnerability disclosure with exploit reuse function $q_1(t) = q_2(t) = 1, \forall t$.

Shellphish plays optimally (Strategic-Shellphish, see Figure 5.7).

Based on the evaluation, we found that Shellphish's score did *not* change when ByteWeight is disabled. This implies that ByteWeight does not contribute Shellphish's performance in the Cyber Grand Challenge. The reason is that, Shellphish's optimal strategy does not involve patching. Although ByteWeight helps to improve patching, it does not change Shellphish's optimal strategy, and thus does not impact the security outcome of the game. This observation highlights that *although techniques advance security skill, they may not improve security outcome for agents who own the techniques*.

## 6.4 Discussion

**Pure-strategy Nash Equilibrium.** In our investigation, we study the exploit reuse technique class and evaluate it on a security model, the autonomous computer security game. In the evaluation, we found that pure-strategy Nash Equilibrium always exists. Although it is known that

Nash Equilibrium always exists for zero-sum games, pure-strategy Nash Equilibrium does not. In the future, we plan to study the observation in a mathematical way and explore whether or in what condition pure-strategy Nash Equilibrium exists for autonomous computer security game.

**More Plot Investigation.** We plan to investigate more cases under different parameters setup. For example, in the technique class evaluation, we found that when both players have the exploit reuse skills $q_1(t) > 0.5$ and $q_2(t) > 0.5$, the areas when both players disclose a vulnerability are separated. One future work is to investigate the reason why the areas are separated and to explore the analytical result for vulnerability disclosure in terms of a technique class such as exploit reuse.

## 6.5 Summary

In this chapter, we introduce qualitative technique evaluation, which is to evaluate techniques from the security outcome's perspective. Different from known evaluations which typically measure techniques on specific skill level, qualitative technique evaluation focuses on the impact of the overall security outcome, such as agent's utility, vulnerability disclosure etc. Qualitative technique evaluation is general for all software security techniques, and it measures an essential question for software security — does a technique make us more secure? In addition, we propose two methods for qualitative technique evaluation, and we apply our methods to study two binary analyses: the technique class of exploit reuse and ByteWeight, a function identification technique. We found that exploit reuse helps to decrease zero-day attacks, and that ByteWeight does not help Shellphish with more scores in the Cyber Grand Challenge final since Shellphish's optimal strategy does not involve the patches generated by the assistance of ByteWeight.

# Part IV

# Conclusion

# Chapter 7

# Conclusion

Software security techniques are essential; they enable the automation of the software analysis process and efficiently protect software programs that we have heavily relied on in our life. However, software security techniques have their merits in practice only to the extent they can be used to achieve a goal. Unfortunately, the software security community has overlooked offense and defense strategy for software vulnerabilities. Due to the lack of study in offense and defense strategy, parties are making suboptimal decisions for software vulnerabilities, and those decisions have resulted in a significant loss. For example, the WannaCry attack is majorly caused by NSA's zero-day vulnerability stockpiling, and that a team such as Shellphish suffered by suboptimal strategy during the Cyber Grand Challenge final competition.

In this dissertation, we propose a *holistic* approach to reason about software security. The approach considers both software security techniques and offense and defense strategy. We introduce the connection between techniques and strategy, and we propose a new methodology to holistically study software security. From the technical perspective, we invent ByteWeight and ShellSwap, two new techniques for function identification and automatic exploit reuse, respectively. From the strategy perspective, we abstract software security as autonomous computer security games, and we show that the autonomous computer security game model can be used for calculating optimal strategies. Finally, we blend techniques and strategy by establishing qualitative technique evaluation, and we investigate the qualitative security impact for ByteWeight and automatic exploit reuse.

# Bibliography

[1] * Prefix idioms seem quite important. * Only looking at gaps, author = Rosenblum, Nathan E. and Zhu, Xiaojin and Miller, Barton P. and Hunt, Karen, booktitle = Proceedings of the 23rd National Conference on Artificial Intelligence, pages = 798–804, publisher = AAAI, title = Learning to Analyze Binary Computer Code, url = http://www.aaai.org/Papers/AAAI/2008/AAAI08-127.pdf, year = 2008.

[2] CVE-2017-0144, Common Vulnerability and Exposures. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0144`.

[3] Cve vulnerabilities by date. `https://www.cvedetails.com/browse-by-date.php`.

[4] Microsoft Security Updates. `https://technet.microsoft.com/en-us/security/bulletins.aspx`.

[5] QEMU. `http://www.qemu-project.org/`.

[6] Timeline: NSA hacking tool to WannaCry. `https://boot13.com/windows/timeline-nsa-hacking-tool-to-wannacry`.

[7] Tracer. `https://github.com/angr/tracer`.

[8] U.S. declares North Korea carried out massive WannaCry cyberattack. `https://www.washingtonpost.com/world/national-security/us-set-to-declare-north-korea-carried-out-massive-wannacry-cyber-attac` `2017/12/18/509deb1c-e446-11e7-a65d-1ac0fd7f097e_story.html?`

noredirect=on&utm_term=.adef7e7d009f.

[9] The wannacry attack. http://malware.wikia.com/wiki/WannaCry.

[10] WannaCry ransomware attack losses could reach 4 billion. https://www.cbsnews.com/news/wannacry-ransomware-attacks-wannacry-virus-losses.

[11] WannaCry Ransomware: What We Know Monday. https://www.npr.org/sections/thetwo-way/2017/05/15/528451534/wannacry-ransomware-what-we-know-monday.

[12] WannaCry: the biggest ransomware attack in history. https://www.raconteur.net/infographics/wannacry-the-biggest-ransomware-attack-in-history.

[13] CGC Final Event File Archive. https://repo.cybergrandchallenge.com/CFE/, 2016.

[14] The CGC Repository. https://github.com/CyberGrandChallenge, 2016.

[15] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.

[16] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity—principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1):1–40, 2009.

[17] American fuzzy lop. http://lcamtuf.coredump.cx/afl.

[18] R. Anderson and T. Moore. The economics of information security. *Science (New York, N.Y.)*, 314(2006):610–613, 2006.

[19] T. Avgerinos, S. K. Cha, B. T. H. Lim, and D. Brumley. AEG: Automatic exploit generation. In *Proceedings of 18th Annual Network and Distributed System Security Symposium*. Internet Society, 2011.

[20] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with

Veritesting. In *Proceedings of the International Conference on Software Engineering*, pages 1083–1094, New York, New York, USA, 2014. ACM Press.

[21] R. Axelrod and R. Iliev. Timing of cyber conflict. *Proceedings of the National Academy of Sciences*, 111(4):1298–1303, 2014.

[22] G. Balakrishnan. *WYSINWYX: What You See Is Not What You Execute*. PhD thesis, University of Wisconsin-Madison, 2007.

[23] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. ByteWeight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium*, pages 845–860. USENIX, 2014.

[24] T. Bao, Y. Shoshitaishvili, R. Wang, C. Kruegel, G. Vigna, and D. Brumley. How shall we play a game:a game-theoretical model for cyber-warfare games. In *IEEE 30th Computer Security Foundations Symposium*, pages 7–21, 2017.

[25] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017.

[26] BAP: Binary analysis platform. `http://bap.ece.cmu.edu/`.

[27] L. Bilge and T. Dumitras. Before we knew it—an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 833–844. ACM, 2012.

[28] BinDiff. `http://www.zynamics.com/bindiff.html`.

[29] BinNavi. `http://www.zynamics.com/binnavi.html`.

[30] BitBlaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`.

[31] M. Bourquin, A. King, and E. Robbins. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM Program Protection and Reverse Engineering*

*Workshop*. ACM, 2013.

[32] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[33] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *IEEE Transactions on Dependable and Secure Computing*, 5(4):224–241, 2008.

[34] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 143–157. IEEE, 2008.

[35] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Network and Distributed System Security Symposium*. The Internet Society, 2010.

[36] H. Cavusoglu, H. Cavusoglu, and S. Raghunathan. Emerging issues in responsible vulnerability disclosure. *Fourth Workshop on the Economics of Information Security*, pages 1–31, 2005.

[37] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[38] S. Choi, H. Park, H.-i. Lim, and T. Han. A static birthmark of binary executables based on API call structure. In *Proceeding of the 12th Asian Computing Science Conference*, pages 2–16. Springer, 2007.

[39] R. A. Clarke and R. K. Knake. *Cyber War: The Next Threat to National Security and What to Do About It*. HarperCollins, 2010.

[40] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by

blocking bad input. In *Proceedings of 21st Symposium on Operating Systems Principles*, pages 117–130. ACM, 2007.

[41] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 133–147. ACM, 2005.

[42] CVE Details. Vulnerabilities By Type. https://www.cvedetails.com/vulnerabilities-by-types.php, 2017.

[43] G. Cybenko and C. E. Landwehr. Security analytics and measurements. *IEEE Security and Privacy*, 10(3):5–8, 2012.

[44] C. Czosseck and K. Podins. A vulnerability-based model of cyber weapons and its implications for cyber conflict. *International Journal of Cyber Warfare and Terrorism*, 2(1):14–26, 2012.

[45] M. Daniel. Heartbleed: Understanding when we disclose cyber vulnerabilities, 2014.

[46] DARPA. Cyber Grand Challenge final statistics. http://www.lungetech.com/cgc-corpus/team/cfe/Shellphish/.

[47] DARPA. Cyber Grand Challenge frequently asked questions. https://cgc.darpa.mil/CGC_FAQ.pdf.

[48] DARPA. DARPA's Cyber Grand Challenge: Final event program. https://www.youtube.com/watch?v=n0kn4mDXY6I.

[49] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou. The complexity of computing a Nash equilibrium. *Society for Industrial and Applied Mathematics*, 39(1):195–259, 2009.

[50] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, N. Stefan, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium*. The Internet Society, 2012.

[51] L. M. Dermed and C. L. Isbell. Solving stochastic games. In *Annual Conference on Neural Information Processing Systems*, pages 1186–1194, 2009.

[52] Dia2dump Sample. http://msdn.microsoft.com/en-us/library/b5ke49f5.aspx.

[53] Dyninst API. http://www.dyninst.org/.

[54] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedins of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 75–88. USENIX, 2006.

[55] IDA FLIRT Technology. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml.

[56] ForAllSecure. Mayhem. https://forallsecure.com/blog/2016/08/06/mayhem-wins-darpa-cgc/.

[57] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An inquiry into the nature and causes of the wealth of internet miscreants. *Proceedings of the 14th ACM conference on Computer and communications security*, pages 375–388, 2007.

[58] B. Fung. The NSA hacks other countries by buying millions of dollars' worth of computer vulnerabilities, 2013.

[59] GCC—Function Inline. http://gcc.gnu.org/onlinedocs/gcc/Inline.html.

[60] A. Greenburgh. Kevin Mitnick, once the world's most wanted hacker, is now selling zero-day exploits, 2014.

[61] I. Guilfanov. Decompilers and beyond. In *BlackHat USA*, 2008. hexray.

[62] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, 2005.

[63] S. Heelan. Automatic generation of control flow hijacking exploits for software vulnera-

bilities, 2009.

[64] C. Herley and P. C. Van Oorschot. Sok: Science, security, and the elusive goal of security as a scientific pursuit. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, 2017.

[65] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Security Symposium*, pages 177–192, 2015.

[66] 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 *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, pages 969–986. IEEE, may 2016.

[67] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. *Proceedings of the Virtual Execution Environments*, pages 2–12, 2006.

[68] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, page 611, New York, New York, USA, 2009. ACM Press.

[69] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories*, pages 329–338. IEEE, 2013.

[70] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 271–286. USENIX, 2004.

[71] J. Kinder. *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, 2010.

[72] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270. USENIX, 2004.

[73] K. Leyton-Brown and Y. Shoham. *Essentials of Game Theory: A Concise Multidisciplinary Introduction*. 2008.

[74] Z. Lin, X. Zhang, and D. Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 247–256. IEEE, 2008.

[75] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Security Symposium*, pages 239–254. USENIX, 2005.

[76] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200. ACM, 2005.

[77] Y. Luo, F. Szidarovszky, Y. Al-Nashif, and S. Hariri. Game theory based network security. *Journal of Information Security*, 1:41–44, 2010.

[78] K. Lye and J. M. Wing. Game strategies in network security. *International Journal of Information Security*, 4(1-2):71–86, 2005.

[79] L. MacDermed, C. L. Isbell, and L. Weiss. Markov games of incomplete information for multi-agent reinforcement learning. In *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*, pages 43–51, 2011.

[80] M. H. Manshaei, Q. Zhu, T. Alpcan, T. Bacar, and J.-P. Hubaux. Game theory meets network security and privacy. *ACM Computing Surveys*, 45(3):1–39, 2013.

[81] T. Moore, A. Friedman, and A. D. Procaccia. Would a 'cyber warrior' protect us? Explor-

ing trade-offs between attack and defense of information systems. In *Proceedings of the Workshop on New Security Paradigms*, pages 85–94, 2010.

[82] T. Muller. Aslr smack and laugh reference seminar on advanced exploitation techniques. Technical report, RWTH Aachen University, 2008.

[83] J. F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36(1):48–49, 1950.

[84] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.

[85] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[86] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 87–102. ACM, 2009.

[87] N. Perlroth and D. E. Sanger. Nations buying as hackers sell flaws in computer code, 2013.

[88] S. L. Pfleeger. Security measurement steps, missteps, and next steps. *IEEE Security and Privacy*, 10(4):5–9, 2012.

[89] S. L. Pfleeger and R. K. Cunningham. Why measuring security is hard. *IEEE Security and Privacy*, 8(4):46–54, 2010.

[90] N. A. Quynh. Unicorn - The ultimate CPU emulator. `http://www.unicorn-engine.org/`.

[91] N. Robert Joyce. Disrupting nation state actors. `https://www.youtube.com/`

watch?v=bDJb8WOJYdA.

[92] A. Romano and D. Engler. Expression reduction from programs in a symbolic binary executor. In *Proceedings of the 20th International Symposium Model Checking Software*, pages 301–319. Springer, 2013.

[93] N. Rosenblum. The new Dyninst code parser: Binary code isn't as simple as it used to be, 2006.

[94] S. Roy, C. Ellis, S. Shiva, D. Dasgupta, V. Shandilya, and Q. Wu. A survey of game theory as applied to network security. In *Proceedings of the 43rd Hawaii International Conference on System Sciences*, pages 1–10. IEEE, 2010.

[95] Samurai CTF Team. We are Samurai CTF and we won Defcon CTF this year. AMA!, 2013.

[96] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[97] H. C. Schramm, D. L. Alderson, W. M. Carlyle, and N. B. Dimitrov. A game theoretic model of strategic conflict in cyberspace. *Military Operations Research*, 19(1):5–17, 2014.

[98] E. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22nd USENIX Security Symposium*, pages 353–368. USENIX, 2013.

[99] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium*, pages 379–394, 2011.

[100] K. Sen, G. Necula, L. Gong, and W. Choi. Multise: multi-path symbolic execution using value summaries. In *Joint Meeting on Foundations of Software Engineering*, pages 842–853, New York, New York, USA, 2015. ACM Press.

[101] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without func-

tion calls (on the x86). In *ACM Conference on Computer and Communications Security*, pages 552–561, 2007.

[102] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. *Proceedings of the 11th ACM conference on Computer and communications security CCS 04*, page 298, 2004.

[103] L. S. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095, 1953.

[104] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 16th Network and Distributed System Security Symposium*. Internet Society, 2008.

[105] Shellphish. Mechanical Phish. `https://github.com/mechaphish`.

[106] Shellphish. Patcherex. `https://github.com/shellphish/patcherex`.

[107] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, pages 138–157. IEEE, 2016.

[108] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using rescue points to navigate software recovery. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 273–280. IEEE, 2007.

[109] A. Smirnov and T.-c. Chiueh. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium*, 2005.

[110] 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 *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[111] M. Tambe, M. Jain, J. A. Pita, and A. X. Jiang. Game theory for security: Key algorithmic

principles, deployed systems, lessons learned. In *Proceedings of the 50th Annual Allerton Conference on Communication, Control, and Computing*, pages 1822–1829. IEEE, 2012.

[112] The MITRE Corporation. CVE: Common Vulnerabilities and Exposures.

[113] The NOPSRUS Team. DEF CON CTF 2007.

[114] Unstrip. `http://www.paradyn.org/html/tools/unstrip.html`.

[115] M. J. Van Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 27–36. IEEE, 2004.

[116] D. A. Wheeler. How to prevent the next Heartbleed.

[117] W. Xu, S. Bhatkar, and S. Brook. Taint-enhanced policy enforcement : A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, pages 121–136, 2004.

[118] K. Zetter. US used zero-day exploits before it had policies for them, 2015.

[119] L. Zettlemoyer, B. Milch, and L. Kaelbling. Multi-agent filtering with infinitely nested beliefs. In *Advances in Neural Information Processing Systems*, pages 1–8, 2009.

[120] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.

[121] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, pages 337–352, 2013.

[122] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *International Conference on Software Testing, Verification and Validation*, pages 421–428, 2010.