The Pennsylvania State University
The Graduate School

EVALUATING THE ATTACK SURFACE OF CONTROL FLOW INTEGRITY

A Dissertation in
Computer Science and Engineering
by
Dongrui Zeng

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2021

The dissertation of Dongrui Zeng was reviewed and approved by the following:

Gang Tan
Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Trent R. Jaeger
Professor of Computer Science and Engineering

Sencun Zhu
Associate Professor of Computer Science and Engineering

Peng Liu
Professor of Cybersecurity
Director, Center for Cyber-Security, Information Privacy, and Trust (LIONS)

Ben Niu
Senior Software Development Engineer at Microsoft
Special Member

Chita R. Das
Professor of Computer Science and Engineering
Department Head of Computer Science and Engineering

Abstract

Control-Flow Integrity (CFI) enforces a control-flow graph (CFG) to limit attackers' ability to manipulate runtime control flow. The essence of a CFI enforcement is a binary-level CFG, which we call a CFI policy. However, there are many CFI variations, each enforcing CFGs of a certain precision level. Each precision level achieves different effectiveness of eliminating attack surface, resulting in different security guarantees. In general, enforcing a more precise CFG exposes a smaller attack surface. However, the remaining attack surface after a CFI defense may leave programs still vulnerable. Therefore, evaluating the attack surface of a CFI-protected software is critical and desired.

The first step is to construct CFGs corresponding to different CFI implementations. Some CFI systems construct CFGs based on binaries alone but others modify compilers to access source-level information for better CFG precision. Extracting CFGs from different CFI implementations can be laborious. Thus, we propose to generate CFGs of different precision levels based on standard compiler-generated meta-information, including symbol tables, relocation information, and debugging information. The key component of the system is a type-inference engine that infers types of low-level storage locations, which enables various signature matching methods for constructing CFGs at different precision levels.

Different CFGs result in different security strengths; the ideal measurement of security strength would be the attack-surface reduction. We define the attack surface as all attack paths that fulfill an attacker's malicious goal. Due to the unavoidable path explosion problem in finding all paths in a program, the key of designing a good metric for the attack surface is to balance the completeness, accuracy, and scalability of the static analysis used for revealing the attack surface. Therefore, we propose two quantitative metrics for the attack surface of a CFI-hardened program, one pursuing the completeness of covering the attack surface with overapproximation while the other one favoring better accuracy. Computing the two metrics requires a program's CFG, an attack model, and a security-violation policy as input. The first one relies on an attack-aware data dependency tracking algorithm to identify all risky program points; and the second one relies on a per-path value tracking analysis to determine risky paths.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First, I want to give my great gratitude to my advisor, Prof. Gang Tan, for his patient mentoring and support for my Ph.D study. I have learned a lot from his expertise and personality. He teaches me how to do rigorously understand issues and solve problems. Without his guidance, I would never have been able to achieve my current progress.

Besides my advisor, I would like to thank the members of my dissertation committee, Prof. Trent Jaeger, Prof. Sencun Zhu, Prof. Peng Liu, and Dr. Ben Niu, for their time and helpful comments.

Also, I was glad and honored to meet and work with my great labmates, Ben Niu, Shen Liu, Robert Brotzman-Smith, Sun Hyoung Kim, Michael Norris, and Yongzhe Huang, during my graduate study. Thanks to their help and company. Last but not the least, I am truly grateful to my family for their continuous support.

This dissertation is based upon work fully/partially supported by US NSF grants CCF-1624124, CNS-1624126, CNS-1801534 and CCF-1723571. The involved research was also supported in part by the Defense Advanced Research Projects Agency (DARPA) under agreement number N6600117C4052, DARPA grant HR0011-19-C-0106, and Office of Naval Research under agreement number N00014-17-1-2539.

Chapter 1
Introduction

In this chapter, we first give an introduction on control-flow hijacking attacks and defenses proposed to mitigate such attacks, where Control-Flow integrity (CFI) is the most practical defense. Enforcing different CFGs by CFI results in different effectiveness of preventing control-flow hijacking attacks. Thus, we then discuss at a high level what factors influence the attack surface of CFI and the challenges of evaluating the attack surface. At the end, we give the thesis statement and summarize our contributions in addressing all challenges.

1.1 Control-Flow Hijacking and Defenses

While C/C++ programs enjoy high performance resulted from the complete programmer control of memory, memory-corruption vulnerabilities, such as out-of-bound accesses, format string manipulation, and use-after-free bugs, are made possible, which undermines the security of programs. Memory-corruption vulnerabilities alone are safety issues; however, when there is an attacker who aims to exploit the vulnerabilities to achieve a malicious goal, such as injecting malicious code to steal private information or compromise the underlying operating system, the safety problem is escalated to a security problem.

Control-flow hijacking is one of the attacking techniques enabled by memory corruptions. The idea is to make use of memory corruptions to manipulate the content of control-flow related memory locations so that the destination of a computed control-flow transfer (i.e., indirect branches) can be controlled by the attacker. By chaining multiple controlled indirect branches, such as a sequence of return instructions with a crafted stack of desired code addresses (i.e., return-

oriented programming), the attacker compromises the program to reach a state where the attacker's goal is fulfilled. For example, the control flow is directed to an `execve` system call to open a shell for the attacker.

Given the difficulty of eliminating all memory-corruption vulnerabilities in a program, researchers resort to defenses that mitigate control-flow hijacking attacks, including Data-Execution Prevention (DEP), Address Space Layout Randomization (ASLR), Stack Canary, and Control-Flow Integrity (CFI). DEP is proposed to prevent control-flow hijacking attacks that result in injecting malicious code (code-injection attacks). However, the attacker can still make use of the existing code to achieve a malicious goal (code-reuse attacks). For example, an attacker can first lead the control flow to an `mprotect` system call to change the protection flags for a memory region and continue with a code-injection attack. Therefore, ASLR is invented to hide the memory layout from the attacker by randomizing the starting addresses of different chunks of data and code. However, ASLR can be compromised by information leaks, heap spraying, and even brute-force guessing. Stack Canary achieves some level of stack protection by monitoring the integrity of a random value injected between stack frames, which is also easy to bypass by guessing the value or an arbitrary memory write primitive.

On top of DEP, ASLR, Stack Canary, CFI is proposed to directly protect indirect branches. The idea is to constrain the target of each indirect branch within a legitimate scope, which limits attackers' ability to manipulate the program's control flow. A CFI scheme consists of two major components: the CFI policy and the CFI enforcement. The CFI policy, which is a binary-level control-flow graph (CFG), defines what control flows are legitimate in a program; and the CFI enforcement makes sure the program's runtime respects the policy. In this thesis, we focus on evaluating the attack surface of CFI policies and assume a perfect runtime enforcement for each policy.

1.2 Attack Surface of Control-Flow Integrity

While it is generally agreed that applying CFI increases a program's security strength, it is hard to evaluate how much strength a particular CFI variation adds. The fundamental difficulty is because what CFI enforces is just a control-flow graph, which does not directly tell what, or how much, security it provides. For instance,

it is unclear whether, or to what extent, the attacker can perform arbitrary code execution, which is a typical attacking goal, after CFI is enforced. In other words, there is a *fundamental gap* between what CFI enforces and what is desired for security, which is the reason why previous work found that programs with a CFI defense can still be exploited [2–10]. Therefore, CFI does not prevent all control-flow hijacking attacks, indicating that there is still attack surface for breaking CFI. Then, we have reached a question: can we evaluate the attack surface of CFI to understand how much security is improved by CFI?

To evaluate the attack surface, the first step is to define what attack surface is. However, attack surface is often used as a conceptual and opaque idea to express the feasibility and possibility of launching attacks; there has not been a precise definition of attack surface. Next, we use prior work's definitions and the drawbacks of the corresponding metrics to motivate our way of defining attack surface. In fact, we have observed that there can be multiple reasonable definitions, each motivating some metrics for quantitatively measuring the attack surface.

One existing way is to represent the attack surface with the available gadgets (e.g., [8, 10, 11]), which we call the *gadget-level abstraction*. A metric following this level of abstraction has a number of drawbacks. First, it depends on a particular gadget-finding tool. Every tool counts particular kinds of gadgets. Some tools count only code snippets ending with return instructions, while other tools also count snippets ending with indirect calls. An incomplete gadget set is insufficient for measuring the whole attack surface. Also, it does not take gadget chaining into account; even if the number of available gadgets is small in a CFI-hardened program, it does not mean good security if there are many ways of chaining gadgets into attacks.

Another kind of abstraction is to define the attack surface based on the outgoing edges of each indirect branch, with the intuition that enforcing a CFG with less edges should reduce the attack surface; we call this the *edge-level abstraction*. Metrics following this definition describe the average freedom of manipulating control flow at indirect branches, which attempts to give an overall evaluation of the *control-flow manipulating space* given a powerful attack model where the target of every indirect branch can be manipulated by the attacker. Such metrics include AIR (Average Indirect target Reduction) [12, 13], the sizes and the number of target sets [14], AIBT (Average Indirect Branch Target) [15], the median and

distribution of target set sizes [16], AICT (Average Indirect Call Target) [17, 18], and runtime indirect branch targets [19]. For example, AICT is calculated as the number of targets averaged over all indirect calls. However, there is still a gap between the control-flow manipulation space and the whole attack surface of CFI, since manipulating indirect branches in most cases is only part of an attack.

Therefore, to accurately measure attack surface, end-to-end attacks should be considered. However, it is impossible to know all attacks ahead of time. We believe the best way is to perform evaluation against a precisely defined security-violation policy, which defines a malicious goal of an attacker. In this way, the attack-surface evaluation can be strictly limited to a specific attacking goal so that the evaluation result would not be over-interpreted. In fact, the control-flow manipulation space is the attack surface for attacks only aiming at tampering the indirect branches during runtime.

We further observe that the attack surface is influenced by the assumed attack model; a powerful attack model shall yield a larger attack surface. The strongest attack model assumed by CFI work is an attacker who resides in a concurrent thread with the victim thread; thus, the memory of the victim thread can be modified in any way and at any time. This attack model is also what the gadget-level and edge-level metrics all assume. However, if we want to understand the attack surface with respect to a weaker attack model (e.g., the attacker can only cause memory corruption at a certain point), the existing gadget-level and edge-level metrics become overapproximations. Therefore, an accurate attack-surface evaluation should be able to adjust to different attack models.

In all, with the obvious facts that CFI is able to limit the attacker's ability to manipulate control flows and the attack surface should be program-dependent, we conclude that the attack surface is influenced by five factors: the **abstraction granularity**, the **attack goal**, the **attack model**, the **CFI policy**, and the **target program**. The abstraction granularity is about what entities are used to represent the attack surface, such as the gadgets used by the gadget-level abstraction and the outgoing edges used by the edge-level abstraction; we call the entities *attack-surface entities*, which will be discussed with more details when we propose our metrics. The attack goal defines what kind of attacks the evaluated attack surface is for. The attack model defines what capability the attacker has in terms of influencing the program's runtime. The CFI policy is the enforced CFG that

protects the target program. Therefore, we define that the evaluation of the attack surface is to discover all attack-surface entities from the target program, with respect to an enforced CFG, a specified attack goal, and an attack model.

1.3 Challenges of Evaluating the Attack Surface

Now, we have summarized the five critical factors that influence the attack surface and defined what attack-surface evaluation is. However, each of the five aspects brings challenges to the goal of evaluating the attack surface.

Attack-surface abstraction. There can be different abstractions of the attack surface. The existing approaches have explored two choices: the gadget-level abstraction and the edge-level abstraction, where the gadgets and edges are the corresponding attack-surface entities (i.e., gadgets and edges are used to represent the attack surface). However, a real control-flow hijacking attack needs to chain multiple gadgets or to manipulate multiple indirect branches. Thus, these two abstractions are simplifications of control flow hijacking attacks; in other words, the two abstractions are coarse-grained. Therefore, a challenge is to design a fine-grained abstraction of attack surface, while considering the complexity of evaluation. For example, it is intuitive to define an ideal evaluation of the attack surface as finding *all real attacks*. However, due to the significant difficulty of identifying a real attack, it is challenging to discover all real attacks to reveal the whole attack surface.

Attack goal. Control-flow hijacking is an attacking technique for an attacker to achieve a malicious goal. However, different security scenarios pose different applicable malicious goals. For instance, a malicious goal towards a private web server can be stealing private information or injecting fake data; however, overloading the server might be less interesting, since the potential gain of such an attack is limited. In contrast, increasing the energy consumption of a data center or overloading a cloud server may become attractive for attackers. The existing gadget-level metric cares about a goal that the attacker aims to identify gadgets without chaining them into attacks; the existing edge-level metrics assume the attacker's goal is to manipulate the destination of an indirect branch. In all, the attack goals considered by existing metrics only reflect initial steps in a control-flow

hijacking attack. Therefore, a challenge is to propose attack goals that are more towards the ultimate purposes of attackers and to represent attack goals in a systematic way.

Attack model. An attack model defines the attacker's capability of influencing the program's runtime, which is an important factor that impacts the discovery of attack-surface entities. For example, the powerful attack model assumed by prior metrics makes it easy to discover all the attack-surface entities; that is, all gadgets and all indirect-branch edges are treated as the attack-surface entities. Note that for a weaker attack model, some of the gadgets and edges may be discharged from the attack surface; in other words, existing metrics would exaggerate the attack surface of a weaker attack model. Therefore, a challenge is to transform descriptive definitions of attack models into a more systematic formalization so that the attacker's influences can be reflected in the discovery of attack-surface entities. Also, we need to propose some weaker attack models to facilitate our attack-surface evaluation. The attack model influences the difficulty of discovering attack-surface entities. For example, if we assume the attacker can only influence a program through legal user-interface and we would like to use real attacks to represent the attack surface, an attack-surface evaluation is the problem of discovering all workable exploits, which is demonstrated to be challenging [20–24].

CFI policy. A CFI policy is the enforced binary-level CFG. The main difficulty of constructing binary-level CFGs is to compute control-flow targets for indirect branches, which include return instructions, indirect jumps (jumps via register or memory operands), and indirect calls (function calls via register or memory operands). Since indirect-branch targets depend on values that are computed at runtime, it is challenging to predict them statically.

Researchers have explored two main approaches for binary-level CFG construction: the binary-analysis approach and the compiler-modification approach. The *binary-analysis* approach [12, 18, 25–27] analyzes binary code and its contained information for CFG construction and has the benefit of not requiring source code. However, CFGs constructed by this approach are of a relatively low precision [28]. The *compiler-modification* approach for binary-level CFG generation [13, 29–31] modifies existing compilers to propagate information from source to binary code and then performs CFG construction directly at the binary level. This approach con-

structs CFGs of much higher precision than the binary-analysis approach. However, this approach relies on source code; and every implementation only corresponds to a specific compiler and a specific version. In all, there is a wide range of CFI schemes (e.g., [12, 13, 25, 27, 29, 32, 33]) and each makes its own choice of CFI policy (i.e., what precision of CFGs it enforces). A comprehensive discussion of CFI variations can be found in [14].

Different CFI schemes may enforce different CFGs. For the goal of evaluating the attack surface of CFI, it is necessary to acquire CFGs that are used by existing CFI work. On the other hand, to understand how CFG precision influences CFI's effectiveness in reducing the attack surface, constructing CFGs of a specific precision level is critical. Given our demand for a variety of CFGs, extracting CFGs from existing CFI implementations is not ideal. Some CFI implementations use the binary-analysis approach while the others employ the compiler-modification approach. The binary-analysis approach may make assumptions about the binary code, such as the compilers used for producing the binary and the instruction set used by the binary. On the other hand, the compiler-modification approach is compiler-specific; each CFI implementation of this sort relies on a specific compiler version. Thus, CFGs extracted from different CFI implementations are not comparable due to their different assumptions. Moreover, there is no prior work capable of generating CFGs according to a specified precision level. In all, a practical challenge is to design a unified CFG construction approach that can produce CFI policies necessary for our purpose of evaluating the attack surface.

1.4 Thesis Statement and Contributions

As we introduced in this chapter, evaluating the attack surface of a CFI-protected program needs to consider multiple aspects, while existing quantitative metrics are coarse-grained. However, given the challenges mentioned above, are there fine-grained quantitative metrics for evaluating the attack surface which can scale to complicated large software?

Thesis Statement. *The attack surface of a CFI-protected program can be measured by fine-grained and scalable quantitative metrics.*

Contributions. This thesis systematically explores the method of evaluating an

attack surface of a CFI-protected program. By following the methodology, we propose two fine-grained quantitative metrics for measuring the attack surface. One relies on an attack-aware dependency tracking analysis to discover risky program points as the abstracted attack-surface entities, which pursues to cover *all* attack-surface entities while improving abstraction granularity over prior metrics; the other one identifies risky paths as the abstracted attack-surface entities, which aims to further increase the abstraction granularity at the cost of incomplete coverage of the attack-surface entities. With the new metrics and our flexible CFG construction approach, we conduct a comprehensive comparison of multiple meaningful metrics and yield a deep understanding in the relation between attack surface and CFG precision. In detail, we make the following contributions:

- **Risky program points as the attack surface.** We present MazeRunner, the first framework that applies an attack-aware dependency tracking to quantitatively measure the attack surface of a CFI-protected program [34]. In this work, the attack surface is represented by risky program points (point-level abstraction), which is more fine-grained than the abstractions of existing metrics. In this work, we propose a formalization of attack goals and attack models. Since the attack goals and attack models considered by MazeRunner are more realistic than the those used by prior metrics, MazeRunner gives a more informative security evaluation of CFI policies. Moreover, it takes both control and data flow into account to better correlate to potential attacks, also making MazeRunner's measurement more fine-grained. Furthermore, we propose a series of optimization techniques to make MazeRunner scalable to large programs.

- **Security application of MazeRunner.** Our attack-aware dependency tracking and the detection of risky program points are overapproximated. Thus, with respect to an attack model and a security-violation policy, when our metric for a program is zero, MazeRunner guarantees that the program is secure. One can use this feature to discharge unnecessary CFI/DFI runtime checks to improve performance.

- **Risky paths as the attack surface.** We propose a new metric based on a path-level abstraction of the attack surface, where paths are the attack-surface entities. The attack model is even weaker than what MazeRunner

assumes; thus, the evaluation result is more realistic and informative. The core of the security determination for each path is a per-path value tracking analysis, which overapproximates attack paths and achieves a 100% precision in detecting safe paths. In all, the new metric further increases the granularity over MazeRunner's metric. We implement a framework called SpaceExplorer to compute the metric.

- **Security application of SpaceExplorer.** The per-path value tracking analysis can be used to improve the efficiency of automatic exploit generation. The analysis can determine safe paths without involving SMT solvers. Thus, it can discharge a large amount of safe paths from the candidates for symbolic execution, which is necessary for the automatic exploit generation.

- **Comprehensive metric comparison** Using our path-level metric, which is the most fine-grained measurement we can compute, as reference, we compute all other applicable metrics and do a comprehensive comparison to demonstrate their advantages and disadvantages. Moreover, we perform a statistical experiment to understand the relation between the attack surface of CFI and the CFG precision enforced by CFI.

- **Flexible CFG construction.** To enable the comprehensive metric comparison, we propose a flexible CFG construction approach. Our CFG construction is the first one that produces CFGs from standard meta-information generated by compilers. Compared to the approach of modifying compilers, most of our system can be reused across compiler versions and across compilers. Compared to the binary-analysis approach, our system is capable of generating CFGs of much higher precision. In addition, our approach designs different signature matching methods to produce CFGs used by different CFI work. Moreover, our approach is able to produce CFGs of a specific precision level.

- **Binary-level type-inference.** The key step in our CFG construction is a general binary-level type-inference procedure. Flow-based constraints are generated from binary code together with meta information and are solved to infer types of registers and stack locations. Our binary-level type-inference engine can be reused for other binary analysis.

1.5 Outline

The rest of the dissertation is organized as following. Chapter 2 introduces necessary background information and related work. Chapter 3 discusses our flexible CFG construction. Chapter 4 elaborates on using risky program points as a metric for the attack surface. Chapter 5 presents another attack-surface metric relying on risky paths. Chapter 6 performs a comprehensive comparison of multiple meaningful attack-surface metrics. Chapter 8 concludes this dissertation and Chapter 7 depicts future directions.

Chapter 2
Related Work

In this section, we discuss the most related research of this dissertation.

Binary-level Type Inference. Classic binary-level type-inference systems mostly assume stripped binaries ([35] provides a comprehensive survey). Such systems rely on constraint solving, machine learning, and heuristics. Some of them support type hints to improve the accuracy of type inference, where type hints may come from the user's expertise and type signatures of standard library calls. The type inference used in our CFG generation system uses debugging information as hints. This comes with the downside of assuming non-stripped binaries, but it enables a more complete type inference since debugging information provides a wealth of source-level types. Thanks to debugging information, our flow-based type inference can infer most of the types in operands of indirect branches, while previous systems on stripped binaries had trouble of inferring function-pointer types [35, 36], even after employing more complex techniques including heuristics (e.g., for classifying whether an immediate is an integer or a pointer), points-to analysis, and dynamic analysis.

Binary-level CFG Construction. Static disassembly of binaries and CFG construction have always been a challenge in reverse engineering binary programs. Other than the standard linear-sweep and recursive-traversal algorithms, other attempts at better static disassembly and CFG construction have been presented in previous papers. Kruegel et al. [37] proposed to combine linear sweep with recursive traversal. Their system first identifies function boundaries using heuristics and then performs recursive traversal within each function. The recursive traversal for a function starts at every byte to accommodate variable-sized instructions in x86;

conflicts between disassembled basic blocks are resolved using a set of heuristics. The system does not follow the control flows of indirect branches and suffers from incomplete disassembly. Balakrishnan et al. [38] proposed Value Set Analysis (VSA) that can sometimes statically determine the control-flow targets of indirect branches. Wartell et al. [39,40] described a machine-learning approach for discovering the most likely disassemblies with the help of a training corpus. TypeArmor [27] performs liveness-analysis based arity matching to refine the control-flow targets of indirect calls. In particular, for a caller, liveness analysis is used to decide whether there is a return value from the callee and the number of arguments that are passed; for a function, it deduces the number of formal parameters and whether the function returns a value to the caller. Then an indirect call is allowed to target any function whose number of parameters match the the number of arguments passed from the indirect call and whose return value information matches the indirect call's. All previous systems assume stripped binaries.

Control-flow integrity. There are many CFI variations. According to the precision of the CFG that is enforced, we can roughly divide CFI work into two categories: *coarse-grained* and *fine-grained*. Coarse-grained CFI implementations (e.g., [12, 25, 26, 41–43]) enforce imprecise CFGs. Fine-grained CFI (e.g, [13, 27, 29–31, 44–49]) enforces fine-grained CFGs in the sense that each indirect branch can have its own target set. Intuitively, fine-grained CFI provides stronger security as it further restricts attackers' freedom in manipulating control flow. However, programs hardened by fine-grained CFI can still be compromised by controlling data that is critical to control flow or through data-only attacks [5–7].

CFI policies also differ in how backward control flow is protected. For example, MCFI [13] treats backward edges the same as forward edges in the CFG: the return instruction of function f can return to any possible caller of f, irregardless of the calling context. A context-sensitive way of checking backward control flow is through a shadow stack (proposed in the original CFI [32]), ensuring that the return instruction of f can return to f's immediate caller in the current calling context. More advanced CFI systems keep track of runtime information to make both forward and backward control-flow checks context-sensitive and sometimes even path-sensitive [11, 48, 50–52].

Another way of protecting control flow is through memory protection. For

example, Code-Pointer Integrity [53] isolates code pointers into a protected memory region and prevents attacks from controlling those code pointers. However, this method is out of scope of this dissertation. Instead, this dissertation focuses on evaluating the security of CFI systems, which protect control flow by inserting checks before indirect branches. Using a path-based method to evaluate the security of memory-protection based control flow protection is left for future work. Also, with the cloud becomes critical in computing, a variation of control-flow integrity is applied to remote attestation [54–65]. The essence of control-flow attestation is still a control-flow policy. For example, [61] stores paths at the server to validate paths reported from clients, while [65] reduces the server's burden of storage by using an Arity CFG produced by TypeArmor [27].

CFI evaluation. In evaluating a CFI, prior systems use the reduction of gadgets or graph-based metrics to justify their security; their limitation has already been discussed in the introduction. Xu et al. [66] propose a set of metrics for measuring a CFI solution's compatibility, applicability, and relevance; it further creates a test suite for testing CFI solutions. However, the work focuses on the CFI implementations but is not designed to evaluate CFI policies through measuring the attack surface as what we do. Muntean et al. [16] employ static taint tracking to approximate the actual indirect branch targets and use the median and distribution of target set sizes to evaluate the control-flow manipulation space; as we explained in introduction, it is not measuring the attack surface we are interested in. Li et al. [19] aim to investigate the difference between a CFI's design and implementation and estimate runtime targets of indirect branches to measure the control-flow manipulation space under a strong attack model. In all, existing CFI evaluation approaches do not consider how attacks can be launched in their security metrics.

Attack Generation. A few advanced systems apply symbolic execution to automatically chain gadgets and search for potential attacks, such as AEG (Automatic Exploit Generation) systems [20–24] and Proof-of-Concept (PoC) attack generation systems [2–10]. For example, BOPC [9] chains basic blocks into a path to achieve a user-specified functionality, such as making an infinite loop or calling a specified library function. An attack goal can be modeled as a functionality for BOPC to work on. Our way of generating risky paths is different. BOPC understands the semantics of each basic block by symbolic execution and chains them together to

form paths according to a template parsed from the specified functionality. However, attack generation systems discover only a small number of attack instances, which is insufficient in revealing the complete attack surface of a CFI-protected program. In contrast, MazeRunner attempts to overapproximate attack surface given a specific attack and defense setup, which is different from prior systems that aim to identify attacks, while SpaceExplorer discovers a large number of paths based on CFG only and uses a per-path value tracking analysis to determine if a path achieves an attack goal without involving symbolic execution.

Chapter 3
Flexible CFG Construction

3.1 Overview

As mentioned earlier, there are various CFG precision levels chosen by different CFI implementations. Each precision level has its own merit. In terms of security, high-precision CFGs are preferred. But as for runtime efficiency, low-precision CFGs are more favored. Also, when source-level information is limited, low-precision CFGs are often used due to the simplicity of construction. Due to the practical difficulty of extracting CFGs from different CFI implementations, we propose a CFG construction approach, capable of generating CFGs of high precision levels, without compiler modifications [67]. Moreover, with simple customization, the framework can flexibly accommodate a range of precision levels, which are critical for performing a large-scale statistical analysis to understand the relation between CFG precision and the size of attack surface.

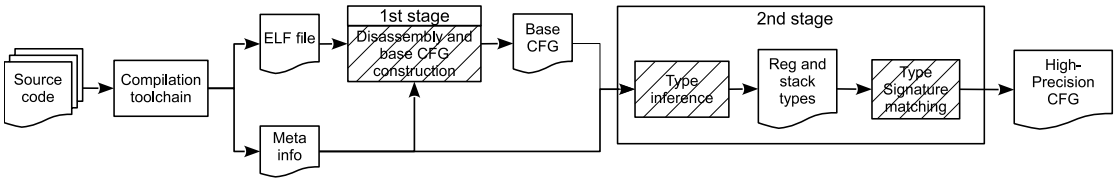


Figure 3.1. System architecture for high-precision CFG construction.

System workflow. Figure 3.1 presents the architecture of our CFG-construction system. The system takes source code, which is fed to an unmodified compilation toolchain (a compiler and a linker) to generate a binary program with standard meta-information. It then constructs the binary program's CFG in two stages.

In the first stage, a recursive-traversal disassembly algorithm is applied to disassemble the binary program and construct a coarse-grained CFG, which is called the base CFG in the figure. The first stage also takes as input compiler-generated meta-information, which tells the entries of all functions and makes disassembly complete. Its generated CFG is coarse grained in that many targets are allowed for indirect branches. On the other hand, control-flow edges out of direct branches (gotos, if conditionals, and direct calls) are accurate because their targets can be statically computed.

The second stage takes the base CFG and the meta information as input and refines the control-flow targets of indirect branches. Its key component is a type-inference engine that infers types of storage locations from types in debugging information. From the inferred types, the second stage follows a type-signature approach [13, 29] to narrow down the targets of each indirect callsite.

Following the two-stage design, we have built a prototype system that constructs high-precision CFGs for binaries that are generated from different compilers (GCC and LLVM) and different compiler versions. The prototype is for Linux x86-32 binaries in the ELF format.[1] Further, the prototype assumes that the source code is in the C language because its type-inference engine infers C-like types for storage locations. Previous work [47,68] has shown that the type-signature approach applies equally well on C++'s type system with the help of Class Hierarchy Analysis (CHA). This work and recent work on C++ CFG construction [69] give us confidence that our meta-information based approach can be generalized to programs in C++, with more work on type inference.

As a high-level note, our approach is compiler independent since the meta information relied on by our system (symbol tables, relocation and debugging information) all have standard formats that are independent of compilers; in particular, ELF and DWARF. As long as a compiler supports those standard formats, the parsing and use of meta information are compiler independent.

[1]The prototype relies on a previous formal model of x86-32 for decoding [1]. That model lacks the decoding support for x86-64.

3.2 Compiler-generated Meta-information

Standard compiler-generated meta-information includes symbol tables, relocation information, and debugging information. Such information is critical for our flexible CFG construction tool. We next briefly describe the information contained in each kind.

Symbol tables. Compilers generate information about symbols (e.g., function and variable names) from source code in the form of symbol tables and embed the tables in binaries. Tools such as linkers and debuggers consume symbol tables to locate and relocate a program's symbolic definitions for static/dynamic linking and debugging. Symbol tables contain entries for symbols and each entry stores a name, a binding address, the type of the symbol, and other information. Our system compiles source code to generate unstripped binaries, which carry the full symbol tables.

Relocation information. Before linking, memory addresses of compilation units such as functions and global data are unknown. Compilers therefore generate relocation entries so that static/dynamic linkers can patch the program's code and data after memory addresses become known. The patch process is for relocating code and data in object code and is crucial for separate compilation.

Debugging information. There are several formats for debugging information including STABS [70] and DWARF [71]. We focus on DWARF since it is the standard format adopted by most compilers including GCC and LLVM and also most debuggers.

The DWARF format is well-designed for debugging and includes several kinds of debugging information: (1) information in source code such as types and scope of identifiers is included in the form of *Debugging Information Entries (DIEs)*; (2) line-number information is included for recording the correspondence between binary and source code; (3) *location descriptions* are included for describing storage locations of variables during execution; (4) the *Canonical Frame Address* (CFA) information is included for describing the stack layout; this information is used during debugging and can also be useful for producing back traces when exceptions are raised. More details about debugging information, especially types, will be

discussed in Section 3.3.1.

Incomplete and inaccurate meta-information. Symbol tables and relocation information are critical to static and dynamic linkers and therefore it is reasonable to assume that they are complete and accurate. The picture is different for debugging information, however. First, while all compilers generate a basic set of debugging information, the generation of advanced debugging information is compiler dependent. For example, GCC generates call-site information, which tells whether calls are tail calls and the types of arguments, while LLVM does not produce such information. Second, debugging information may be inaccurate, especially in optimized code. It is well known that line-number information becomes unusable in optimized code: after optimizations such as code motion, it is often not possible for the compiler to update the relationship between binary and source code in terms of line numbers. Another example is that the CFA information for describing stack layouts may become inaccurate after compiler optimizations.

3.3 Type Inference

The key component in our CFG-building framework is a generic type-inference system that infers types of storage locations (registers and stack slots) from source-level types included in debugging information. The type-inference engine works on one function at a time, starting from types in debugging information (which tell at least the entry types of parameters and local variables) and inferring the types of storage locations in the middle of the function.

It has four major components: (1) a component that collects types from debugging information; (2) a stack-layout inference algorithm that normalizes the representation of stack slots with respect to a canonical frame address; (3) a constraint-generation component that turns instructions and control flows into type constraints; (4) a constraint-solving component that solves constraints and computes a set of types for storage locations at every code address in the function. We next discuss these components in detail.

```
    // Assume variable i is initially at the top of the stack
addr1:  ...
        ...
addr2:  eax := mem(esp)
        ...
addr3:  ebx := eax
        ...
addr4:  ret
```

Debugging entry for `i` :
int; `[(CFA-8,[addr1,addr2]), (eax,[addr2+1,addr3]), (ebx,[addr3+1,addr4])]`

Figure 3.2. An example for debugging entries.

3.3.1 Debugging type information

Type inference starts from those types already included in debugging information. We next describe in detail what type information is there in debugging information. At a high level, types of source-code identifiers are included. What types of information are attached depends on the kinds of identifiers.

Functions. For a function, debugging information contains a debugging entry for the function, followed by entries for the function's formal parameters and local variables. From the entry of the function and the entries of its formal parameters, we can know the function's number of parameters, the parameter types, the return type, and also the range of code addresses of the function's body.

Formal parameters and local variables. These are function-local identifiers. The debugging entry for such an identifier contains the type of the identifier as well as a location description that describes where the identifier is stored.

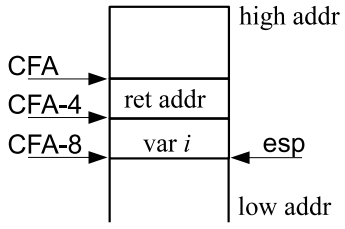
A location description $[(st_1, rng_1), (st_2, rng_2), \ldots, (st_n, rng_n)]$ is a list of pairs, where st_i is a storage location, which is either a register (such as `eax` in x86-32), a stack location, or a location in global data sections (which store global variables), and rng_i is a code-address range. The above location-description list is interpreted as follows: the identifier is stored in location st_i before any instruction whose address is within address-range rng_i.

Figure 3.2 presents an example. In this and other examples of the chapter,

for clarity we will use a pseudo-assembly syntax whose notation is described as follows. We use "`:=`" for an assignment (that is, a move instruction). When an operand represents a memory operand, the memory address is put into `mem(-)`. For instance, `mem(esp)` is a memory operand with the address in `esp`, while `esp` itself represents a register operand. We will also use the syntax "`if ... goto ...`" for conditional jumps (on x86, it represents a comparison followed by a jump instruction).

The example in Figure 3.2 includes the skeleton code for a function. Assume there is a local variable named `i` in the function and it is initially allocated on the stack; specifically, `i` is stored at the top of the stack. At `addr2`, the storage location of `i` is moved to register `eax` and at `addr3` the location is moved to `ebx`.[2] The debugging entry for i is also shown in the figure (for brevity, we have omitted information irrelevant for our discussion, such as line numbers). It tells the source-code type of the variable and its location description, which means that the variable is stored in stack slot "`CFA`-8" between `addr1` and `addr2`, in `eax` between `addr2+1` and `addr3`, in `ebx` between `addr3+1` and `addr4`.

The representation of a stack slot is normalized relative to `CFA` (Canonical Frame Address). The `CFA` address is typically defined to be the value of the stack pointer at the call site that invokes the current function. Therefore, the return address is stored at "`CFA`-4" since the return address is pushed by the call instruction (and the stack grows from high to low memory addresses). When local variables are allocated on the stack, their addresses are expressed relative to `CFA` in location descriptions. For the example, the relation between `CFA` and `esp` at the beginning is shown as follows:



The types and location descriptions in debugging entries for local variables (and parameters) tell the types of some storage locations at specific code addresses. For

[2]In unoptimized code, a local variable is stored in a specific stack slot for the entire function; however, code produced by higher-optimization levels can move the storage of a variable to a register to improve the efficiency of accessing the variable.

the example in Figure 3.2, the debugging entry for `i` tells that int is the type of stack slot `cfa-8` between `addr1` and `addr2`, the type of `eax` between `addr2+1` and `addr3`, and the type of `ebx` between `addr3+1` and `addr4`.

Global variables. Similar to local-variable entries, debugging entries for global variables tell their types and location descriptions. However, a global variable's storage location does not change over the course of program execution and therefore its location description has only one pair $[(st, rng)]$, where st encodes the memory address where the global variable is stored in global data sections, and rng is either the entire range of code addresses or a portion of it depending on whether the global variable is external or static.

Incomplete type information. As we have shown, debugging information tells the types of some storage locations at certain code addresses. However, types in debugging information are for source-level constructs and, when source code is compiled to binary code, compilers are conservative in embedding types in debugging information, resulting in incomplete type information. As an example in Figure 3.2, the location description does not tell the type of `eax` between `addr3+1` and `addr4`, even though the type should remain the same before `eax` is modified. As another example, suppose a local variable is stored in `eax` initially and is then copied to `ebx`, and the location description states that the variable is stored in `eax` even after copying; then `ebx`'s type after copying should be the same as `eax`, even though debugging information does not tell it directly. Section 3.5 will show that for large programs types can be missing in debugging information for over 50% of indirect-call operands.

3.3.2 Stack layout inference

As presented earlier, stack slots in debugging information are normalized and encoded as offsets to `CFA`. A binary program accesses stack slots, however, via memory addresses in the form of offsets to registers such as `esp` and `ebp`. To be able to track and infer types of stack slots, we must infer the relationship between registers and `CFA`. As a simple example, suppose the program reads the stack at address "`esp`" and debugging information tells us that the value at stack slot "`CFA-8`" is of type t; we cannot know the type of the value at address "`esp`" unless it is

```
      {esp=CFA-4}
1   push ebp
      {esp=CFA-8}
2   ebp := esp
      {esp=CFA-8, ebp=CFA-8}
3   push ebx
      {esp=CFA-12, ebp=CFA-8}
4   esp := esp - 8
      {esp=CFA-20, ebp=CFA-8}
5   if  eax==0 goto 7
      {esp=CFA-20, ebp=CFA-8}
6   ebx := esp
      {esp=CFA-20, ebp=CFA-8, ebx=CFA-20}

      {esp=CFA-20, ebp=CFA-8}
7   esp := esp + 8
      {esp=CFA-12, ebp=CFA-8}
8   pop ebx
      {esp=CFA-8, ebp=CFA-8}
9   pop ebp
      {esp=CFA-4}
10  ret
```

Figure 3.3. Stack layout inference for a toy function.

given that `esp=CFA-8`.

Some compilers generate in debugging information call-frame information that encodes the relation between `CFA` and register values. However, it is often incomplete and sometimes inaccurate. For instance, LLVM generates call-frame information for function prologues but such information is not generated for code after prologues. GCC's call-frame information may become inaccurate after code is optimized, for example, when code is moved after return instructions during optimization. Therefore, we have built a static analysis that infers the relation between registers and `CFA`; we call it stack layout inference.

The static analysis takes the assembly code of a function and its base CFG as input and at every address builds a set of equations of the following form: $\{r_1 = \texttt{CFA} + \mathit{off}_1, ..., r_n = \texttt{CFA} + \mathit{off}_n\}$. If a register does not point to the stack, then no equation for the register is included in the above set.

At the beginning of the function, the analysis assumes "{esp=CFA-4}" (the four bytes between CFA and esp are the return address). The transfer function for an instruction is built straightforwardly based on the instruction's semantics. Figure 3.3 provides some examples: after "push ebp", we have "{esp=CFA-8}"; after another instruction "ebp:=esp", we get "{esp=CFA-8, ebp=CFA-8}".

At a control-flow merge point, the merge function is defined to be the intersection of the sets of equations from the paths being merged. In Figure 3.3, address 7 is both a destination of the if-test at address 5 and the fall-through destination of the instruction at address 6. Therefore, we take the intersection of "{esp=CFA-20, ebp=CFA-8}" and "{esp=CFA-20, ebp=CFA-8, ebx=CFA-20}".

For a function call, the stack layout remains the same after the call; it assumes a function call preserves the stack layout. If the function has a loop, a standard work-list algorithm is used to calculate a fixed point; the work-list algorithm terminates since the underlying lattice is of finite height.

With the result of stack layout inference, accesses to the stack via registers plus constants can be normalized with respect to CFA.

3.3.3 Constraint generation

At a high level, constraints specify relations between types of storage locations based on how values flow in the input function. Our system tracks the types of registers, stack slots, and locations in global data sections. It does not explicitly track the types of heap locations that are dynamically allocated; however, values read from memory can still be assigned types in some cases. For instance, if eax holds a pointer to a dynamically allocated struct and debugging information tells that eax's type is a struct-pointer type, then a memory read via eax plus a constant offset returns a value whose type can be inferred from the struct type and the offset.

Figure 3.4 presents the syntax of type constraints. We use *st* for a storage location, which is either a register or a stack slot in the form of CFA plus some offset. We use *gl* for the location of a global variable (which resides in global data sections). We separate *gl* from *st* since the type of a global variable does not change over the course of program execution (further, debugging information tells the type); in contrast, types in registers and stack slots may change over program execution.

$$
\begin{array}{rcl}
st & := & r \mid \texttt{CFA} + \mathit{off} \\
x, y, z & := & st_l^- \mid st_l^+ \mid gl \\
C & := & x \supseteq y \mid x \supseteq *(y + \mathit{off}) \mid x \supseteq \&(y + \mathit{off}) \\
 & & \\
t & := & \top \mid \text{void} \mid \text{int} \mid \text{char} \mid \text{float} \mid \text{double} \mid t* \mid t[n] \mid \\
 & & (t_1, \ldots, t_n) \to t \mid \{id_1 : t_1, \ldots, id_n : t_n\} \mid \text{tbl\_ent} \\
X, Y, Z & := & x \mid x : t \\
D & := & X \supseteq Y \mid X \supseteq *(Y + \mathit{off}) \mid X \supseteq \&(Y + \mathit{off})
\end{array}
$$

Figure 3.4. Syntax of type constraints.

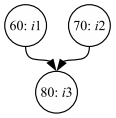
Code	Constraints
`10: eax:=ebx`	$\texttt{eax}_{10}^+ \supseteq \texttt{ebx}_{10}^-$ $\forall st \neq \texttt{eax},\ st_{10}^+ \supseteq st_{10}^-$
// assume `ebp=CFA-8` `20: eax:=mem(ebp+12)`	$\texttt{eax}_{20}^+ \supseteq (\texttt{CFA} + 4)_{20}^-$ $\forall st \neq \texttt{eax},\ st_{20}^+ \supseteq st_{20}^-$
// `ebx` unrelated to `CFA` `30: eax:=mem(ebx+4)`	$\texttt{eax}_{30}^+ \supseteq *(\texttt{ebx}_{30}^- + 4)$ $\forall st \neq \texttt{eax},\ st_{30}^+ \supseteq st_{30}^-$
// assume `ebp=CFA-8` `40: eax:=lea(ebp+12)`	$\texttt{eax}_{40}^+ \supseteq \&(\texttt{CFA} + 4)_{40}^-$ $\forall st \neq \texttt{eax},\ st_{40}^+ \supseteq st_{40}^-$
// `ebx` unrelated to `CFA` `50: eax:=lea(ebx+4)`	$\texttt{eax}_{50}^+ \supseteq \&(\texttt{ebx}_{50}^- + 4)$ $\forall st \neq \texttt{eax},\ st_{50}^+ \supseteq st_{50}^-$
	$\forall st,\ st_{80}^- \supseteq st_{60}^+$ $\forall st,\ st_{80}^- \supseteq st_{70}^+$

Figure 3.5. Examples for constraint generation.

We use x, y, and z for type variables. They can stand for the set of types of registers or stack slots at a particular point, or the type of a global variable. In particular, we use st_l^- to stand for the type set of storage location st before the instruction at address l, and st_l^+ for the type set of storage location st after the instruction at address l. For instance, $\texttt{eax}_{10}^-$ stands for the type set of `eax` before instruction at address 10 and $\texttt{eax}_{10}^+$ for the type set of `eax` afterwards. Since the type of a global variable does not change, gl is used to stand for the type of the corresponding global variable.

Type constraints are of three forms. Value-flow constraint "$x \supseteq y$" models the

case when y's value flows to x; as a result, x's type set should be a superset of y's type set. A dereference-flow constraint "$x \supseteq *(y + \mathit{off})$" models the case when the value at address $y + \mathit{off}$ is read from memory and flows to x; address $y + \mathit{off}$ is assumed to not point to a stack slot, since stack slots are represented using `CFA` plus an offset. An address-flow constraint "$x \supseteq \&(y + \mathit{off})$" models the case when the memory address of $y + \mathit{off}$ flows to x.

Constraints are generated conservatively from instructions. Due to space limitation, we cannot enumerate all the cases. Instead, we just discuss the cases for typical instructions and illustrate them via examples in Figure 3.5. For a register-to-register move instruction $r_1 := r_2$, constraints are generated to express that the type set of r_1 afterwards is a superset of r_2 beforehand (since r_2's value flows to r_1) and the type set of other storage locations afterwards is a superset of the same storage locations beforehand since their values are unchanged (for brevity, we will omit the mentioning of constraints concerning unchanged storage locations in the rest of the discussion). An example of "`eax:=ebx`" is in Figure 3.5.

For a memory-to-register move $r_1 := \mathit{mem}(\mathit{op})$, constraint generation depends on operand op. Suppose $\mathit{op} = r_2 + \mathit{off}$. If r_2 points to the stack before the move instruction (determined by stack layout analysis), then a constraint is generated to state that r_1's type set afterwards is a superset of the type set of the corresponding stack slot beforehand. An example of "`eax:=mem(ebp+12)`" is in Figure 3.5. If r_2 does not point to the stack, then a dereference-flow constraint is generated to relate r_1 and r_2. An example of "`eax:=mem(ebx+4)`" is included in Figure 3.5. When op is an immediate value, our system checks whether it is a memory address that holds a global variable. If it is, a value-flow constraint states that r_1's type set is a superset of the global variable's type. If it is not, no constraint is generated for r_1 after the move; this is an approximation, an unconstrained type variable will be assigned with the $\top$ type, meaning no information is there for its type. In x86, an operand op can also use other complex addressing modes (such as a base register plus a register times a scalar and a displacement); in these cases, we approximate by not generating constraints for r_1.

x86 also has `lea` (load effective address) instructions, which move memory addresses but do not perform memory reads. Figure 3.5 shows two examples. In example one, "`eax:=lea(ebp+12)`" moves the address of `ebp+12` into `eax`. Assuming `ebp=CFA-8`, the instruction moves the address of stack storage location `CFA+4`

to `eax`; we use an address-flow constraint to model that. The second example is similar, except it moves a heap address; it is also modeled by an address-flow constraint.

Constraints generated for a register-to-memory move $mem(op) := r_1$ also depend on the shape of op. If $op = r_2 + \mathit{off}$, and r_2 points to the stack, then a value-flow constraint is generated to relate the type sets of r_1 and the associated stack slot. Otherwise, no constraints are generated since types of global variables do not change and we do not infer types of heap memory locations.

For a function call, constraints are generated to state that the stack slots are unchanged and values of certain register values are preserved according to the calling convention and the type signature of the callee.

Finally, constraints are also generated based on the control-flow graph. When there is a control-flow edge from instruction $i1$ to instruction $i2$, then the value of a storage location after $i1$ conceptually flows to the storage location before $i2$. The last row in Figure 3.5 illustrates the generation of constraints from an example CFG.

In the next step, constraints are decorated with types that are included in debugging information. Figure 3.4 shows the syntax of types. Type t can be either common base types, a function type "$(t_1, \ldots, t_n) \to t$" whose parameter types are t_1 to t_n and return type is t, a pointer type $t*$, a fixed-size array type $t[n]$, or a struct type in the form of a list of fields and field types. We augment C's type system with type tbl_ent, which is used as the type of jump-table entries and helps our system refine targets of indirect jumps. When debugging information tells that the type of x is t, we write $x : t$. For instance, "$\texttt{eax}^{-}_{10}$: int" means that `eax` before address 10 is of type int. We use capital letters X, Y, and Z for *decorated type variables*, which are either x or $x : t$; the second form is used when debugging information tells that x is of type t. A type constraint C is then turned into a *decorated type constraint* D by annotating type variables in C with types provided in debugging information.

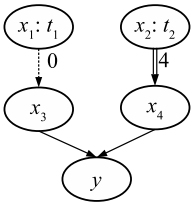
3.3.4 Constraint solving

Decorated type constraints are solved in two steps: (1) constraints are turned into a graph whose nodes are decorated type variables and whose edges model value flows;

(a) Constraints

$x_3 \supseteq \&(x_1 : t_1)$
$x_4 \supseteq *((x_2 : t_2) + 4)$
$y \supseteq x_3$
$y \supseteq x_4$

(b) Constraint graph



(c) Solved constraint graph

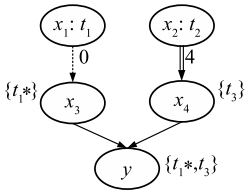


Figure 3.6. An example for illustrating constraint solving. Assume $t_1 = \text{int}$, $t_2 = \{\text{f1} : \text{int}, \text{f2} : \text{float}*\}$, and $t_3 = \text{float}*$.

(2) types are propagated on the graph and a type set for each node is produced.

From constraints to a constraint graph. In the resulting graph, nodes are decorated type variables and edges are of three kinds. For a constraint of the form $X \supseteq Y$, we add a value-flow edge from node Y to node X, written as $Y \rightarrow X$. For a constraint of the form $X \supseteq *(Y + \mathit{off})$, we add a dereference-flow edge from node Y to X and label the edge with *off*; this is written as $Y \overset{\mathit{off}}{\Rightarrow} X$. For a constraint of the form $X \supseteq \&(Y + \mathit{off})$, we add an address-flow edge from node Y to X and label the edge with *off*; this is written as $Y \overset{\mathit{off}}{\dashrightarrow} X$. An example is shown in Figure 3.6: part (a) shows a set of constraints and part (b) shows its corresponding graph.

Constraint graph solving. Algorithm 1 presents the algorithm for solving a constraint graph. At a high level, it uses a worklist to propagate types forward along edges in the graph and computes a *typeOf* function that assigns sets of types to nodes. In more detail, $\mathit{typeOf}(n)$ is initialized according to n's kind: if n is $x : t$, then its type is t since we trust debugging information; if n has no incoming edges, we initialize its type as $\top$, meaning there is no information about the type; otherwise, it is initialized to be $\emptyset$. Then a while loop is used to process the nodes in the worklist. For a node, the algorithm iterates through its outgoing edges. For a value-flow edge $n \rightarrow n'$, when $n' = y$, the types for n are propagated to n'; note if "$n' = y : t$", then there is no need to perform forward propagation since the type of y is already known. For a dereference-flow edge $n \overset{\mathit{off}}{\Rightarrow} n'$, when $n' = y$, the type is propagated according to each type t in $\mathit{typeOf}(n)$. For an address-flow edge $n \overset{\mathit{off}}{\dashrightarrow} n'$, when $n' = y$, the types for n are transformed into correspondent pointer types and propagated to n'. In the case when t is a struct type, we use $\text{offsetTy}(t, \mathit{off})$ to compute the field type in the struct according to a static offset

Algorithm 1 Constraint solving

Global:
$worklist : \mathcal{P}(N)$
$typeOf : N \to \mathcal{P}(Type)$

procedure SOLVE($G = (N, E)$)
  $worklist \leftarrow N$
  for $n \in N$ **do**
    switch n **do**
      case $n =$ "$x : t$":
        $typeOf(n) \leftarrow \{t\}$
      case $n = x$ and n has no incoming edges:
        $typeOf(n) \leftarrow \{\top\}$
      case $n = x$ and n has incoming edges:
        $typeOf(n) \leftarrow \emptyset$
  while *worklist* is not empty **do**
    $n \leftarrow$ remove a node from *worklist*
    for $e \in$ outgoing edges of n **do**
      switch e **do**
        case $e = n \to y$:
          for $t \in typeOf(n)$ **do** ADD(t, y)
        case $e = n \overset{off}{\Rightarrow} y$:
          for $t \in typeOf(n)$ **do**
            switch t **do**
              case $t = \{id_1 : t_1, \ldots, id_n : t_n\}$:
                ADD(offsetTy(t, off), y)
              case $t = t'*$ or $t'[k]$: ADD(t', y)
              case others: ADD($\top$,y)
        case $e = n \overset{off}{\dashrightarrow} y$:
          for $t \in typeOf(n)$ **do**
            switch t **do**
              case $t = \{id_1 : t_1, \ldots, id_n : t_n\}$:
                ADD(offsetTy(t, off)$*$, y)
              case others: ADD($t*, y$)

procedure ADD(t,y)
  if $t \notin typeOf(y)$ **then**
    $typeOf(y) \leftarrow typeOf(y) \cup \{t\}$
    $worklist \leftarrow worklist \cup \{y\}$

for both kinds of edges. Part (c) in Figure 3.6 shows the solved example according to the algorithm.

3.4 CFG Construction

In this section, we present how our system constructs high-precision CFGs for binaries in two stages. The first stage produces a coarse-grained base CFG. The second stage uses type-inference results to enhance CFG precision. In addition, with the type information, we can customize the indirect-branch matching method to produce multiple CFG precision levels.

3.4.1 Base CFG construction

The base CFG construction relies on the classic recursive-traversal disassembly algorithm, which mixes disassembly and CFG construction. It maintains a work list, initialized with known code entries of binaries. The control flows of already disassembled instructions are followed for finding new code addresses to add to the work list for continuing disassembly. It can accommodate data embedded in code since such data should not be targets of control-transfer instructions. However, when a basic block ends with an indirect branch such as an indirect call through a memory operand, without further information it is difficult to predict what the branch may target and add appropriate addresses for further disassembly. A tool may ignore the issue by not adding any address, but this would result in incomplete disassembly. Many tools such as IDAPro make assumptions about compilers and rely on heuristics and code patterns for remedying the situation partially, but they still cannot achieve complete disassembly in all cases since it is limited by the amount of information in binary code.

Our implementation of recursive traversal utilizes meta-information generated by compilers to determine control-flow targets of indirect branches and adds those targets to the worklist for further disassembly. Since it retrieves all possible targets for indirect branches from meta-information, it is able to achieve a complete disassembly and a sound base CFG for further refinement in the second stage of CFG construction.

How our recursive traversal algorithm determines the targets of indirect branches depends on the types of indirect branches. For both indirect calls and indirect jumps, we take advantage of relocation information to narrow down potential targets; therefore, we first explain how potential targets for indirect calls/jumps

are retrieved from relocation information.

Indirect targets retrieval via relocation information. Since a function cannot be invoked indirectly unless its address is taken somewhere in code, several previous systems [13, 25, 32] refine CFGs to allow indirect calls to target only address-taken functions. This is possible because at compile time the compiler does not know the exact addresses of functions and it must generate relocation entries for places where function addresses are used so that during linking they can be patched when the exact function addresses become known. The same applies to targets of indirect jumps; relocation entries are generated at places where those targets are used.

Therefore, our system searches for relocation entries that require the linker to put in absolute addresses during linking. If the symbol name of such an entry is a function name, the function's address must be taken and it is collected into the set FSA (Function-Start Addresses), which is the set of start addresses of address-taken functions. If the symbol name of such an entry is not a function name, the corresponding address is internal to a function and possibly the target of an indirect jump; the address is then collected into ICA (Internal Code Addresses); that is, it is the set of code addresses that can be the targets of indirect jumps.

Handling indirect calls. In the base CFG, an indirect call is allowed to target any address in FSA, the set of start addresses of address-taken functions; furthermore, since functions contained in dynamically linked libraries are called through entries in the Procedure Linkage Table (PLT) in the binary, our system also adds the start addresses of PLT entries to the targets of indirect calls.

Handling indirect jumps. An indirect jump is allowed to target an address if (1) the address is in FSA, or if (2) the address is in ICA and the address falls within the code-address range of the function that contains the indirect jump. The reason for (1) is that tail calls are implemented via indirect jumps; an indirect jump for implementing a tail call is like an indirect call and is therefore allowed to target any function whose address is taken. Note that based on debugging information, we know whether the binary has passed the tail call optimization. If there is no tail call, we do not need to allow FSA to be the targets. For (2), the justification is that non-tail-call indirect jumps are for compiling switch statements and goto statements through labels that are stored in composite data structures; in these

cases, targets of an indirect jump are local (within the function that contains the indirect jump).

Handling returns. When processing a basic block that ends with a direct or indirect call instruction, our recursive-traversal algorithm adds the code address immediately following the call instruction into the disassembly worklist. This is speculative disassembly since it assumes the callee function has a return instruction that will return to the address following the call instruction. The speculative disassembly can produce spurious basic blocks, because the call may target functions that never return; for example, calling the library function `exit` stops the program. Our system assumes that compilers do not put non-executable bytes after call instructions; consequently, spurious basic blocks are not harmful.

With speculative disassembly for return addresses, the targets of return instructions can be computed after the disassembly is complete. Specifically, after computing targets of indirect calls/jumps, our system computes a call graph to compute targets of return instructions. The call graph collects a list of functions that a call instruction can reach, either directly or indirectly through a series of tail calls. Then, return instructions in this list of functions can target the return address following the call instruction.

3.4.2 Type-based CFG construction

Base CFG construction results in coarse-grained CFGs: all indirect calls share the same set of targets; indirect jumps are allowed to target either function-start addresses or some local targets or both. In this section, we discuss how to use the results of type inference described in Section 3.3 to substantially enhance the CFG precision. A previous CFI system [13] adopts a type-signature approach for CFG construction: an indirect call through an operand is allowed to invoke any function whose type is compatible with the operand's function-pointer type. This type-based method requires that source code does not contain type casts that involve function-pointer types; if this is violated, a small amount of changes can be made to the source code to respect the requirement.

The type-signature method requires knowing two pieces of knowledge: (1) the types of functions; and (2) the function-pointer types of operands used in indirect calls. As presented before, types of functions can be acquired from debugging

information. However, debugging information does not always tell the types of operands in indirect calls; this is where our type inference comes into play. After type inference, an operand in an indirect call is assigned a set of types. For each function-pointer type $(t_1, \ldots, t_n \to t)*$ in the set, we allow an indirect call via the operand to invoke any function with type "$t_1, \ldots, t_n \to t$". In the worst case, the set may contain $\top$; for example, when the operand is loaded from a piece of memory for which there is no type information; in this case, the target-set of the indirect call cannot be refined.

For an indirect jump, thanks to the new type tbl_ent, the type-inference also tells whether it is a tail call or a local jump based on jump tables. If the type set of the indirect jump's operand contains only function-pointer types, then it is treated as a tail call and its target set is refined as if it were an indirect call. If the type set of the operand contains only tbl_ent, the set of targets is refined to include only the local targets. Each PLT entry also contains an indirect jump; we allow it to target the address of the dynamic linker as well as a symbolic target that indicates that the jump is allowed to jump into a dynamically linked library. The target sets of remaining indirect jumps are not refined.

After the target sets of indirect calls and jumps are refined, a refined call graph is constructed. The refined call graph is then used to calculate the targets of return instructions.

3.5 Implementation and evaluation

Our system is implemented with a decoder for x86-32, 18K lines of OCaml code for disassembly, type inference, and CFG construction, 4K lines of C code for collecting debugging information, and a few Python and Shell scripts for relocation information collection and data handling.

The disassembler is built based on the decoder. The disassembler first parses the ELF file to obtain all headers and sections. Symbol tables are already retrieved during ELF parsing. Relocation information is collected during compilation with the help of the Linux utility `readelf` and a Python script. Debugging information is collected with the help of the `libdwarf` library, implemented in C. Disassembly and CFG construction are implemented in OCaml and communicate with the debugging-information collection in C via the OCaml-C interface.

For evaluation, we are interested in the following questions: (1) how effective is our type inference for inferring types of operands in indirect branches? (2) How precise are the CFGs generated by our system and how does the precision compare with the binary-analysis approach and the compiler-modification approach?

To answer these questions, we conducted our experiments in a Linux box running x64 Ubuntu 14.0.4 on a PC with 16GB-memory and Intel Core i5-4590 CPU at 3.30GHz. Our evaluation was performed on SPEC2006 C benchmarks and Nginx-1.4.0. Since the type-based approach requires that source code does not contain type casts that involve function-pointer types, we used MCFI's patched SPEC2006 C benchmarks and Nginx-1.4.0 (downloaded from `https://github.com/mcfi`); MCFI made small changes to the benchmarks to remove type casts that involve function pointers (mostly by adding function wrappers). Our prototype system can construct CFGs for both GCC and LLVM at all optimization levels (from `O0` to `O3`). We also tested the CFG construction on multiple versions of the two compilers (in particular, GCC 4.8.4, GCC 5.4.0, GCC 6.2.0, LLVM 3.9, LLVM 4.0). For conciseness, we will present the detailed results only for GCC 4.8.4 and results are generally similar for other GCC versions and LLVM.

Benchmarks	NeedInfer/ Total	ICALL-Precise	ICALL-Imprecise	Inferred Rate	TailCall	Local	PLT	IJUMP-Imprecise
`bzip2`	20/20	20	0	100.0%	0	2	22	0
`sjeng`	1/1	1	0	100.0%	0	15	38	0
`milc`	0/4	0	0	N.A.	0	5	40	0
`sphinx3`	9/9	9	0	100.0%	0	1	60	0
`hmmer`	0/9	0	0	N.A.	1	24	69	0
`h264ref`	40/352	40	0	100.0%	0	12	44	0
`perlbench`	55/109	52	3	94.5%	30	80	114	0
`gobmk`	30/44	30	0	100.0%	0	13	49	0
`gcc`	292/442	291	1	99.7%	20	426	72	1
Total	447/990	443	4	99.1%	51	578	508	1

Table 3.1. Indirect-call and Indirect-jump type-inference results (GCC 4.8.4 with O2 optimization).

3.5.1 Effectiveness of type inference

Table 3.1 presents indirect-branch type-inference results for SPEC2006 C benchmarks. In the table, the benchmarks are sorted according to their sizes in the ascending order. We omitted small benchmarks including `lbm`, `mcf` and `libquantum`

since they do not contain indirect calls/jumps. For each benchmark, column Need-Infer/Total lists the number of indirect calls whose operands do not have debugging type information versus the total number of indirect calls. The numbers show that there are many indirect calls (around 50% for large benchmarks including `perlbench`, `gobmk`, and `gcc`) for which debugging information misses types for their operands.

The second half of Table 3.1 presents indirect-jump type-inference results. Column TailCall presents the number of indirect jumps our system infers as tail calls (recall that the operand type of such an indirect jump should be a function-pointer type). Column Local presents the number of indirect jumps that are inferred as jumps via jump tables. A large number of indirect jumps are in PLT entries and their numbers are shown in Column PLT. Column IJUMP-Imprecise shows the number of indirect jumps whose operand types include $\top$. The data shows that there is only one indirect jump for which our system cannot infer precise information.

3.5.2 CFG precision and validation

For a control-flow graph, Average Indirect-Branch Targets (AIBT) introduced by Niu [72] is defined to be the number of targets averaged over all indirect branches in the graph.[3] The smaller the AIBT is, the more precise the CFG is. In Table 3.2, we present AIBT numbers for SPEC2006 C benchmarks for GCC 4.8.4 across different optimization levels. Each data entry shows AIBTs for the enhanced CFG (left) and the base CFG (right).

Comparison with binary-analysis approach. The precision of base CFGs is roughly the same as the precision of CFGs produced by a typical binary-analysis approach, which commonly allows all indirect calls to target the same set of functions and employs optimizations for handling indirect jumps. As the table shows, the improvement of our type-inference based CFG enhancement is small for small benchmarks; since `lbm`, `mcf`, and `libquantum` do not contain indirect branches, there is no improvement during CFG enhancement. However, for large benchmarks including `perlbench`, `gobmk`, and `gcc`, the CFG improvement is substantial: average

[3]We did not use the AIR (Average Indirect-target Reduction) metric because it has been criticized to not capture the ability of a CFG to withstand control-flow hijacking attacks [7].

Benchmarks	O0	O1	O2	O3	AvgRed
`lbm`	1.7/1.7	1.7/1.7	1.7/1.7	1.7/1.7	0%
`mcf`	1.6/1.6	1.6/1.6	1.5/1.5	1.4/1.4	0%
`libquantum`	3.0/3.0	3.2/3.2	4.0/4.0	4.8/4.8	0%
`bzip2`	2.6/3.0	2.7/3.0	2.5/2.7	1.9/2.2	11.1%
`sjeng`	5.0/9.0	4.9/4.9	6.5/6.5	6.8/6.8	11.1%
`milc`	3.3/3.3	3.3/3.3	3.6/3.6	3.9/3.9	0%
`sphinx3`	4.5/4.6	4.7/4.7	4.9/4.9	5.7/5.7	0.5%
`hmmer`	4.1/7.8	4.8/4.9	4.9/5.0	4.9/4.9	12.9%
`h264ref`	4.0/31.8	4.3/32.1	6.1/34.7	6.2/34.9	84.7%
`perlbench`	17.2/563.0	19.5/102.8	23.6/232.6	34.1/357.6	89.6%
`gobmk`	19.7/104.6	22.0/61.2	19.5/58.1	28.9/68.7	67.4%
`gcc`	23.2/1735.0	26.2/192.8	32.3/212.3	42.3/345.5	89.4%

Table 3.2. Average Indirect-Branch Targets for SPEC benchmarks (GCC 4.8.4).

reduction is between 60 to 90%; many spurious edges were removed during type-based CFG enhancement.

Comparison with compiler-modification approach. When compared with the compiler-modification approach, the precision of our system's enhanced CFGs is only slightly worse than MCFI. A direct comparison by the AIBT statistics is infeasible, because MCFI's implementation does not support 32-bit targets while our current implementation only supports 32-bit binaries. However, based on a fact that both systems use the type-signature approach for matching indirect branches and targets, we can still perform an indirect comparison. MCFI propagates types inside the compiler to binaries; so an indirect call's operand has exactly one type. Our system performs type inference to infer the types of indirect-call operands. As seen in a previous table, around 99% of indirect-call operands get one type signature. Therefore, the CFG precision of our system is very close to the one of MCFI. As a previous survey paper [14] shows, MCFI enforces the highest-precision CFGs among all existing CFI enforcement tools. For instance, Forward-Edge CFI [29] enforces a less precise CFG: it uses arity matching for pairing indirect calls and functions (i.e., it matches the number of parameters) and it does not enforce CFI on return instructions.

Experiment on Nginx 1.4.0. To further evaluate our approach, we conducted

an experiment on Nginx. In Table 3.3, we present experimental results for MCFI-patched Nginx-1.4.0. As the table shows, our approach is able to generate high-precision CFGs for practical applications such as Nginx.

Benchmarks	NeedInfer/ Total	ICALL- Precise	ICALL- Imprecise	Inferred Rate	TailCall	Local	PLT	IJUMP- Imprecise	AIBT	Red
`nginx`	201/289	191	10	95.0%	26	60	133	0	20.7/213.0	90.3%

Table 3.3. Experimental results for Nginx-1.4.0 (GCC 4.8.4 with O2 optimization).

CFG validation. We have performed testing to validate the enhanced CFGs our system generated for SPEC benchmarks and Nginx. We used PIN [73] to instrument the benchmarks' binaries to collect runtime traces for reference data sets and configurations; the reference data sets are included in the original benchmarks and provide good test coverage. The control-flow edges made by indirect branches in the runtime traces were checked to see if they were included in the CFGs generated by our system. In our experiments, CFGs generated by our system for all benchmarks passed the validation. Because the runtime overhead introduced by PIN is substantial and the trace output file is large, we had to perform optimizations for large benchmarks during validation. Since the difficulty of CFG construction lies in control transfers from indirect branches, we instrumented only indirect branches through PIN instead of all control-flow transfer instructions. One downside of resorting to testing for CFG validation is that it can find bugs but cannot show correctness. Unfortunately, the CFI field lacks a method for verifying the correctness of CFGs; addressing this problem would be an interesting research direction.

3.6 Multi-level CFG Construction

With rich source-level information, the matching of indirect branches and functions can be designed to produce CFGs of different precision levels. In other words, the type-signature matching process in the second stage can be customized. For example, we can directly use the C types for matching. As another example, we may want to only match indirect branches and functions by the arity of function arguments. In all, different usages of source-level information lead to different CFG precision levels. So far, we provide 3 classic precision levels: Address-Taken CFG [12, 32], Arity CFG [27, 29], and C-type CFG [13].

On top of the three classic precision levels, we can construct more alternative CFGs of precision levels between the C-Type CFG and the Address-Taken CFG, for the purpose of understanding the relation between CFG precision and attack surface. Such CFGs must cover the C-Type CFG so that the soundness is maintained. That is, the target set of every indirect call needs to cover the targets of the same indirect call in the C-Type CFG. Each alternative CFG is then generated with respect to a specified number of average indirect call targets (AICT), which must be larger than the C-Type CFG's AICT. On top of the C-Type CFG, we need to know the number of uncovered targets to be added to each indirect call, where uncovered targets are those address-taken functions not covered by the C-Type CFG. To decide the number, we first compute the distance of the specified AICT from the C-Type CFG's AICT as well as the distance between the C-Type CFG's AICT and the Address-Taken CFG's AICT. Then, we compute the ratio of the two distances, which tells the percentage of uncovered targets to be added to the indirect call. The selection of uncovered targets is randomized to provide different variations of the same input AICT number.

3.7 Summary

This chapter presents an alternative approach for high-precision CFG construction, without compiler modification. The approach uses compiler-generated meta-information to retrieve source-level information for CFG construction. It relies on a type-inference engine that deduces types of indirect-branch operands from source-level types in debugging information.

The content of this chapter is based on our paper published in CODASPY 2018 [15]. The major difference is that this chapter adds an approach to generating CFGs of multiple precision levels (Section 3.6).

Chapter 4
Risky Program Points as Attack Surface

4.1 Overview

In this chapter, we present a metric relying on an attack-aware dependency tracking analysis to identify risky program points in a CFG [34]. A risky program point is where an attacker can initiate an attack to achieve a malicious goal. Risky program points and the CFG are used to compute a metric for the remaining attack surface. We call the system MazeRunner.

4.1.1 A Motivating Example

Designing a more precise metric to measure the attack surface of a CFI-protected program is critical for understanding the security strength of a CFI policy. We present a high-level example for illustrating the key concepts of measuring the attack surface of a CFI-protected application, which is more accurate than traditional methods.

Security-violation policies. Since we are interested in measuring the attack surface, we use a security-violation policy to model a type of attacks (i.e., when security is violated). This is in contrast to a security policy, which tells when security is preserved; intuitively, the negation of a security policy is a security-violation policy. In our work, A security-violation policy captures a critical state in

a common attack. [1] When this critical state is reached, we say security is defeated and an attack becomes possible. Also, in AEG and PoC attack generation papers, having a list of target functions as the goal for breaking security is an unavoidable design, which is similar to our security-violation policies.

For illustration, we present an example policy that models a critical state in a real attack [2]. This attack has two steps. To bypass memory protection, the first step is to change the protection of a memory region to be both writable and executable; this can be achieved by using ROP attacks to invoke the `mprotect` system call in Linux or `VirtualProtect` in Windows with the right arguments. The second step of the attack is to inject malicious code to the writable and executable region and transfer the control to the malicious code. In this attack, the crucial step is the one that makes a region writable and executable; this step can be reused in many other attacks.

We next present in detail how this crucial step can be modeled in a security-violation policy for Linux; modeling it in Windows would be similar. In Linux, programs compiled by modern compilers do not invoke system calls directly; they instead invoke libc's wrapper functions for system calls. For example, libc has an `mprotect` wrapper, which takes three arguments: the starting address of a memory region, the length of the region, and a protection flag. Therefore, the attacker's goal is to call this libc function with the right arguments to make a region both writable and executable.

To model this in a policy, let us assume that the libc wrapper function for `mprotect` is located at address $libc\_mprot$. Given this we can formally write down the policy, which consists of two parts:

$$\begin{array}{ll} \text{Target:} & \text{pc} = libc\_mprot \\ \text{Attack condition:} & arg_3 \ \& \ 0x6 \ = \ 0x6 \end{array}$$

The policy first tells where security might be violated: when the program counter (pc) is at $libc\_mprot$. It also states an attack condition: when the second-to-last bit and the third-to-last bit in the third argument are both one (those are the bits for the executable and writable protection bits); we use & for the bit-wise and operation. Where arguments can be found is specific to architecture and

[1] Schneider made a distinction between security and safety policies [74]; more precisely, our security-violation policies are limited to safety-violation policies since they are about single states.

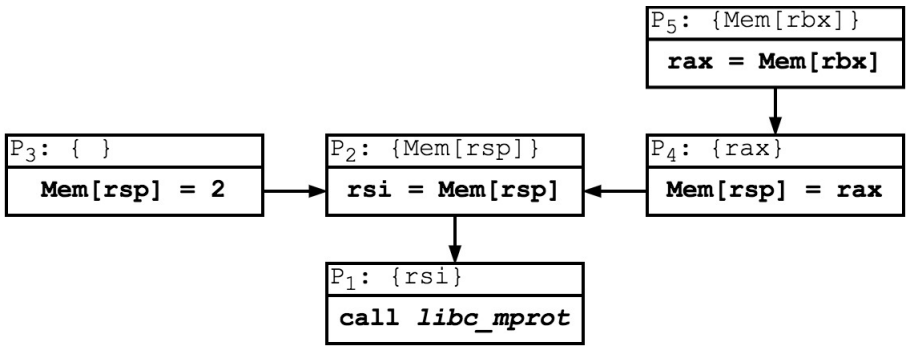


Figure 4.1. Example dependency tracking.

calling convention. In 64-bit x86 Linux, the third argument is in rsi. In 32-bit x86, arguments are passed on the stack in most calling conventions; for instance, the third argument is on the stack at address esp + 16.

Attack-aware dependency tracking. With a given policy, the next step is to perform attack-aware dependency tracking (abbreviated as ADT) along all control-flow paths to identify risky program points where attacks may be launched. Starting from the target specified in the policy, the tracking process uses the attack condition (also in the policy) to infer the initial critical storage locations; it then performs ADT along each path to compute, for each point in the path, a set of storage locations that can be used to launch an attack to cause a security violation at the target location (i.e., to make the security-violation policy satisfied), if those storage locations are controlled by the attacker. For the `mprotect` policy, the initial critical storage location is the third argument.

Since ADT takes a CFG as an input and the CFG must be followed by the input program's control flow because of CFI, it can determine how the target can be reached. For the example policy, the target *libc_mprot* is a libc function and must be reached through a call site. Here, a call site may be either an indirect call (i.e., a call through a register or memory operand) or a direct call. [2] Figure 4.1 presents a toy CFG for the example policy. The CFG contains a call site to *libc_mprot* and our ADT starts with the initial critical storage location rsi (the third argument in x86-64). The tracking result is added to each program point. For example, at P_5 the dependency tracking shows that `Mem[rbx]` may influence rsi at P_1.

[2]When dynamic linking is used, the direct call actually targets *libc_mprot*'s PLT entry, which then invokes *libc_mprot*.

Attack surface calculation. Intuitively, the attack surface of a program should contain two parts: (1) the set of program points where the attacker can initiate an attack, and (2) all paths through which the attack can happen. Therefore, based on the inferred dependency sets and the given attack model about when and what the attacker can influence, MazeRunner first classifies each program point into two kinds: (1) safe, meaning that the attacker cannot make the security violation happen with the given attack model; (2) risky, meaning that the attacker may launch an attack at this point by influencing the storage locations in the tracked dependency set. Then, MazeRunner counts the paths according to the CFG. In all, we would like to define the number of paths that connect any risky program point to the target location as the metric for the attack surface. This metric represents the total risk for a program being influenced to cause a security violation; a larger value means a larger attack surface.

We use Figure 4.1 to illustrate the classification process and our metric computation. We first consider an attacker who can manipulate the heap, but not the stack nor any register. Program point 1 (P_1) is safe, since rsi cannot be influenced at this point. Program point 2 (P_2) is safe, because the dependency set contains an uninfluenceable stack location, assuming rsp points to stack. Program point 3 (P_3) is safe, because the dependency set contains only a constant. Program point 4 (P_4) is safe for the same reason as P_1. But program point 5 (P_5) is risky because rax is assigned from a memory location on the heap, assuming rbx points to a heap location. In total, there is only one risky program point with one feasible attack path; we have 1 as the measurement. In comparison, if we change the attack model so that the attacker can control both the heap and the stack, only P_1 is still safe and other points become risky [3]. In all, there are 4 risky program points, each with one attack path; so, we have 4 as the measurement.

An intuitive comparison with other metrics. With our metric, we can perform a more precise security evaluation for different CFI mechanisms than previous work. We use another toy CFG in Figure 4.2 to explain the advantage of our metric over previous metrics. Suppose the program in Figure 4.1 is part of function `foo` and there exists another function `foobar`, which indirectly calls `foo`. Thus, the attacker can launch an attack from `foobar`. For simplicity, we assume `foobar`

[3]When a program point is determined to be risky, its predecessors are overapproximated to be risky. In this example, P3 and P4 are risky because P2 is risky.

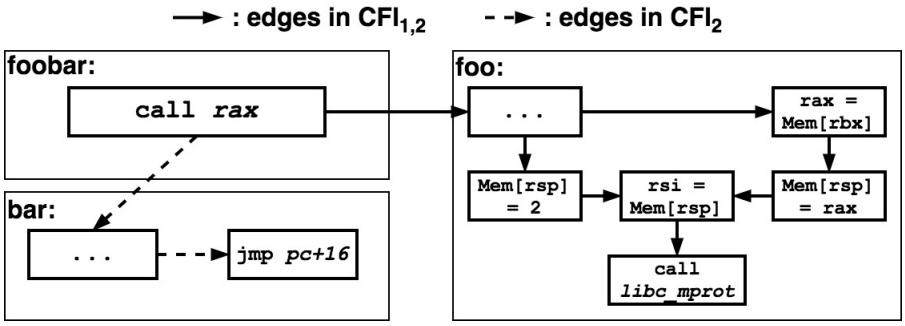


Figure 4.2. Example for metric comparison.

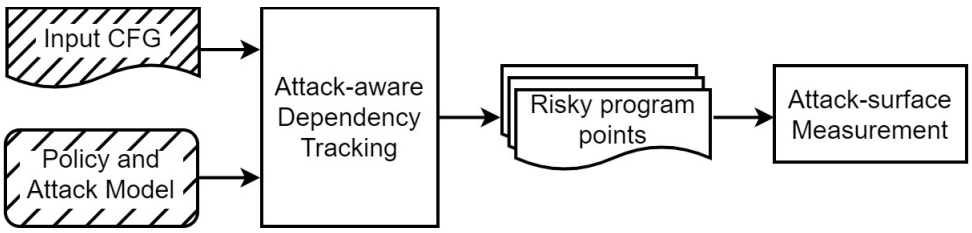


Figure 4.3. MazeRunner's workflow.

contains only the indirect call instruction; and we also assume two different sound CFI implementations, where the indirect call is allowed to target only `foo` in CFI_1 but both `foo` and `bar` in CFI_2. In this case, graph-based metrics would determine that the CFI_1 is safer than CFI_2, since CFI_1 has less edges. However, if the goal of the attacker is to launch an attack according to the `mprotect` policy, neither the additional gadget in `bar` nor the extra freedom of calling `bar` gives the attacker more freedom to launch attacks; that is, the attack surface of the program is not enlarged. In contrast, MazeRunner would compute the same attack-surface measurement for both CFI implementations; we believe it gives a more precise comparison in terms of security.

4.2 System Workflow and Input Specification

System workflow. MazeRunner's workflow is presented in Figure 4.3. MazeRunner takes three inputs: a binary-level CFG, an attack model, and a security-violation policy. The input program and its CFI protection are abstracted as a CFG. The security-violation policy defines what kind of attacks we are evaluating against. The attack model specifies the attacker's capabilities of manipulating control

flow and memory. The attack-aware dependency tracking (ADT) then performs graph exploration to classify program points. From the number of risky program points, MazeRunner computes a metric to quantify the attack surface. As a result, MazeRunner's attack-surface measurement for a CFI-protected program depends on the capability of the assumed attacker as well as the predefined scope of attacks, which better corresponds to the intuition of how to measure an attack surface. We next discuss the three inputs in detail and leave the discussion of MazeRunner's dependency analysis to later sections.

4.2.1 CFGs

A CFG is a static approximation of a program's legal control flow. A binary-level CFG consists of basic blocks of assembly instructions and edges between basic blocks. We say a CFG is *sound* if it includes all the control flow that can happen at runtime during the program's execution. Because a CFG is a static approximation, a single program can have multiple sound CFGs, each of its own precision. For instance, a CFG that allows an indirect call to target every possible code address is sound, but is extremely imprecise. To accommodate different CFGs, we design MazeRunner to take a CFG as input so that it is independent from CFG generation. Our multi-level CFG construction discussed in Section 3.6 is used for our evaluation of MazeRunner.

4.2.2 Attack models

Our framework is parameterized by an attack model, which is about what damage attackers can cause and at what time. Next, we discuss attackers' capabilities of causing damage in terms of control-flow and memory capabilities.

Control-flow capabilities. MazeRunner assumes that attackers have to follow the input CFG; that is, the input program is hardened via CFI so that violation of the CFG is impossible. In other words, all control-flow transfers represented by control-flow edges in a CFG are possible during runtime. Further restriction on backward control flow can be added to ensure that returns match calls, as we will cover when discussing memory capabilities.

Memory capabilities. We split the writable data memory into a stack region

and a heap-global region. The stack region contains stack frames allocated during function calls. The heap-global region contains the heap, which holds dynamically allocated memory and writable global data. In terms of attackers' capabilities to modify memory regions, there are two dimensions: *when* a memory region can be modified and *what* in a memory region can be modified. We represent these capabilities by a labeling system.

On the *when* dimension, we introduce two levels of attacker capabilities:

- **notime**. A region with this label cannot be directly modified by the attacker at any time. For instance, if the stack region has the **notime** label, the attacker cannot directly change the stack. This can be achieved by memory protection or a full-fledged shadow stack [44], which ensures that data on the stack cannot be modified.

- **memwrite**. The attacker can modify the memory region only by memory writes (including system calls that may write to memory). That is, we assume the region changes arbitrarily after a memory-write instruction, but stays the same after other instructions. In real attacks, memory corruptions are done by memory write.

On the *what* dimension, we also introduce two levels:

- **all**. The attacker can modify all locations in the region. In addition, specially for stack, we assume the function parameters cannot be directly manipulated, which can be justified when parameters are passed by registers. For the stack region, it means the top stack frame and callers' stack frames.

- **all-ret** (all except return addresses). This label means that return addresses cannot be modified. For example, all-ret means that the attacker can modify everything on the stack *except* the return address. This assumes a defense for protecting return addresses, e.g., a shadow stack for only return addresses.

The combination of the aforementioned labels enables the specification of a variety of attack models. Among possible attack models, MazeRunner focuses on three attack models:

- **AM0**: (memwrite, all) for stack and heap-global;

- **AM1**: (memwrite, all-ret) for stack and (memwrite, all) for heap-global;
- **AM2**: (notime, _) for stack and (memwrite, all) for heap-global.

For instance, AM1 allows the attacker to modify the stack via memory writes, except for the parameters and return addresses. In Section 4.3.4, we will show that the model (memwrite, all) allows MazeRunner to avoid tracking dependency through the heap-global region, greatly increasing its scalability but still maintaining the precision for comparing different CFI designs. Note that since the two labels of *when* both allow the attacker to launch attacks via any memory write as well as library calls that may write to memory, we can safely limit the scope of our evaluation to application code only. In other words, the attackers following such attack models can launch attacks without compromising the libraries.

4.2.3 Security-violation policies

As discussed, measuring the attack surface against a predefined scope of attacks is necessary. We use security-violation policy to constrain the scope, which captures a critical state in a common attack. We support two genres of policies: general policies and concrete policies. General policies are defined for every indirect call site; controlling the targets and parameters of indirect calls is crucial for a control-flow hijacking attack. So, for a general policy, the target is every indirect call and the attack condition is about the function pointer or the parameters used in the call. General policies are designed for achieving a comprehensive attack-surface evaluation. In contrast, for a concrete policy, the critical state refers to the state before a dangerous function such as `mprotect`. That is, the target refers to a dangerous function and the attack condition describes conditions for arguments required to defeat security.

Before presenting a list of policies supported by MazeRunner, we discuss notation and a representation issue for pointer parameters. First, we use arg_i for the ith argument so that the policy is independent from calling conventions. Second, we introduce a predicate AttackerCtrl(st), meaning that the storage location *st*'s value is sourced from an attacker controlled memory region along a control-flow path. We next present a set of representative security-violation policies for demonstration. The targets in these policies are often used in data-oriented and control-flow hijacking attacks.

- **G1: Target Control**. In this general policy, the target is every indirect call site and the attack condition is when the indirect call operand *op* can be controlled by the attacker. Its attack condition is AttackerCtrl(op).

- **G2: Parameter Control**. This general policy is about the parameters of indirect calls. A general control-flow hijacking attack against CFI needs to channel data flows through function parameters. The attack condition is defined as AttackerCtrl(arg_i) for all i.

- **C1: wx-mem**. In this concrete policy, the target is either `mprotect` or `mmap`. These functions can make a memory region both writable and executable by setting the third argument arg_3 to satisfy $arg_3 \ \& \ 0x6 \ = \ 0x6$.

- **C2: execve**. The target is `execve`. When its first parameter is manipulated, it has the ability of executing any executable in a child process. We define the attack condition to be AttackerCtrl(arg_1).

In addition, we provide a list of candidate policies to show that incorporating new policies is a straightforward process:

- **C3: dlopen**. The first parameter of `dlopen` specifies the name of a shared library to be loaded. If it can be controlled by the attacker, she can use the loaded library to search for critical ROP gadgets or dangerous functionalities. We define the attack condition to be AttackerCtrl(arg_1).

- **C4: system**. Function `system` has only one parameter, which specifies a shell command to be executed. If this parameter is controlled, the attacker can call the "system" library function with an arbitrary shell command. We define the attack condition to be AttackerCtrl(arg_1).

- **C5: write**. Function `write` has three parameters with the first one specifying the output file descriptor, the second one being a pointer pointing to the data source buffer, and the last one limiting the number of bytes that can be written. The three parameters all may harm the security if controlled. For example, when any one of the parameters is controlled, local secret information may be leaked through an attacker-specified information channel. So, the initial dependent storage location can be **any one of the parameters**. Similar policies can be defined over other write-family functions.

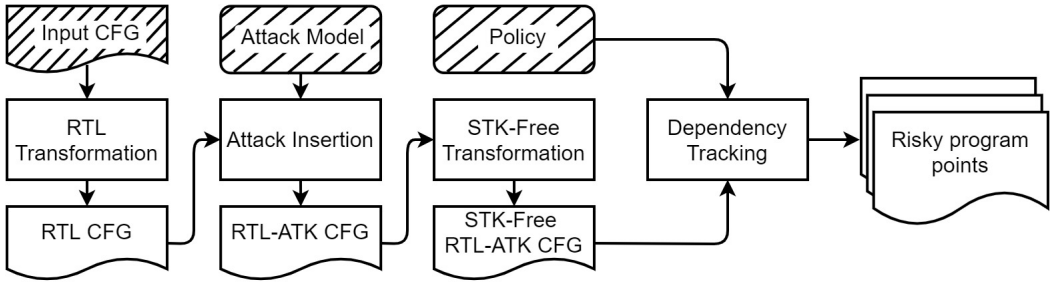


Figure 4.4. Attack-aware dependency tracking steps.

- **C6: read**. Function `read` is similar with `write`, which has three parameters. The first one is a file descriptor for the data source; the second one is buffer pointer for storing read-in data; and the last one specifies the size of data that is expected. Such parameters can be utilized by an attacker for leaking important data as well as injecting unwanted data. Therefore, the initial dependent storage location can be **any one of the parameters**. Similar policies can be defined over other read-family functions.

4.3 Attack-Aware Dependency Tracking (ADT)

The goal of ADT is to track data dependency for the critical state of the specified security-violation policy and identify safe and risky program points. For that, it goes through multiple steps to achieve the goals of correctness, generality, and scalability. Figure 4.4 shows the major steps in ADT. It first converts the input x86 program into an intermediate representation called RTL (RTL Transformation), and inserts attack instructions into the program according to the attack model (Attack Insertion). After that, it simplifies the program by converting it into a stack-free, nondeterministic CFG (STK-Free Transformation). These three steps are our major optimization techniques to address the scalability challenge. Finally, attack-aware dependency tracking is performed on the stack-free CFG to explore risky program points. We next discuss each step in detail.

4.3.1 Conversion into RTL

The first step is to convert the input program into an intermediate representation, in particular, by translating a CFG of x86 assembly instructions into a CFG based on

$$
\begin{array}{rcl}
v & \in & \text{Integers} \\
sz & \in & \text{Integers} \\
regs & := & \text{EAX} \mid \text{EBX} \mid \ldots \\
\mathit{flags} & := & \text{CF} \mid \text{ZF} \mid \ldots \\
loc & := & \text{PC} \mid regs \mid \mathit{flags} \mid \ldots \\
bvop & := & \text{add} \mid \text{and} \mid \text{shl} \mid \ldots \\
cmp & := & \text{lt} \mid \text{eq} \mid \text{gt} \mid \ldots \\
exp & := & \text{bitvec}(sz,\ v) \mid \text{arith}(bvop,\ exp,\ exp) \\
 & \mid & \text{test}(cmp,\ exp,\ exp) \mid \text{ite}(exp,\ exp,\ exp) \\
 & \mid & \text{load\_loc}(loc) \ \mid \ \text{load\_mem}(exp) \\
instr & := & loc = exp \ \mid \ \text{Mem}[exp] = exp \\
 & \mid & \text{IF } exp \text{ DO } instr
\end{array}
$$

Figure 4.5. The major syntax of RTL [1].

RTL (register transfer list) [1]. The RTL (Register Transfer List) language is a small RISC-like language, with a small set of orthogonal instructions and operational semantics formalized in Coq. It was designed for specifying the semantics of assembly instructions: the semantics of an assembly instruction is specified by translating it into a sequence of RTL instructions. Morrisett et al. [1] used the RTL to specify the semantics of x86-32 and MIPS.

Figure 4.5 presents the major syntax of the RTL language. In RTL, values are bit-vectors of certain sizes (e.g., 1 bit, 32 bits, etc.). RTL locations include registers and CPU flags, each holding a bitvector of a certain size. RTL expressions are pure and produce values when evaluated. They include basic bit-vector computations and comparisons, conditional evaluations (ite, if-then-else), and expressions for loading from locations and memory. RTL instructions may modify the state and include assignments to locations, memory writes, and conditional instructions (IF_DO_).

4.3.2 Inserting attack instructions

Since the attacker can modify the program's state during program execution, we model the attacker's changes to the state by inserting attack instructions into the program. We introduce two attack instructions to accommodate our attack models: (1) hgATK represents an attack to the heap-global region; it changes the whole heap and writable global data sections to contain arbitrary values; (2) "sATK$_\alpha$" represents a stack-manipulation attack where the attacker can change the stack; α

specifies what stack elements can be changed, which is one of the what-dimension labels discussed earlier in Section 4.2.2 (i.e., can be all and all-ret).

Where and how these attack instructions are inserted depend on the attack model: the "when" dimension of the attack model determines where attack instructions are inserted in the program; the "what" dimension determines what attack instructions are inserted. As a simple example, if the attack model is (memwrite, all) for the heap-global region and (notime, _) for the stack region, then a hgATK instruction is inserted after every memory-write instruction in the original program and no "sATK$_\alpha$" instruction is inserted.

4.3.3 Conversion to a stack-free representation

Modeling memory is notoriously difficult. We observe many memory accesses in a program are to the stack for accessing local variables and parameters (when parameters are passed on the stack). Therefore, our goal is to convert stack slots, which hold parameters, local variables, return addresses etc., into variables [4], and convert stack accesses into possibly nondeterministic variable accesses. The conversion to a stack-free representation is performed in two steps. First, a *point-to-stack analysis* (PSA) is performed to track pointers to the stack. In fact, PSA is an advanced version of the stack layout analysis (Section 3.3.2) used in the flexible CFG construction component. Based on the result of PSA, we transform the input CFG into a stack-free, nondeterministic CFG. In this way, stack accesses are transformed into variable accesses, simplifying the dependency tracking.

Point-to-stack analysis. PSA takes a CFG as input and performs a lightweight value-set analysis (VSA [38]). Traditional VSA is an interprocedural algorithm that tracks all pointers and numeric values, while our PSA tracks only pointers to the stack and is intraprocedural. In more detail, PSA tracks pointer values in registers and abstract locations (alocs). One aloc represents some memory storage. Each stack frame is partitioned into multiple alocs. We use VSA's heuristic for creating stack alocs: the stack frame of a function is broken into alocs according to the constant offsets used in the function. For instance, if a function has a stack-allocated array and a loop that iterates over the array elements from the beginning of the array, then the whole array is abstracted into one aloc to simplify

[4] More precisely, variables in the RTL language are really RTL locations.

the analysis of pointer arithmetic.

For each function, the analysis starts with the initial fact stating that ESP points to the stack with offset 0, which represents the base of the function's stack frame. Then the intraprocedural VSA algorithm is run on the function's body to compute a set of offsets to the stack base, for registers and alocs and at every program point. For instance, if the computed set for EAX contains offset 4 at a program point, it implies that EAX can hold a pointer to the stack and the pointer has offset 4 to the base of the frame. Note that, due to overapproximation, the computed value set may be $\top$, meaning that the pointer offsets are arbitrary.

To determine the target of a memory access, value sets in registers are sufficient, because all memory accesses in RTL are performed through registers with offsets. If the computed value set for a register at a program point is empty, we can determine that any memory access through that register at the program point is a non-stack access. Otherwise, the memory access is to the stack. Since in x86 the stack goes from high addresses to low addresses, a memory access via a pointer with a negative offset to the frame base is a local-variable access, while an access via a pointer with a positive offset is an access to a function parameter.

Function calls and returns add some complication. During a function call, pointers to the caller's frame may be passed to the callee and used for memory accesses in the callee. Furthermore, the callee can return those pointers to the caller and the caller then uses them to access its stack frame. Our PSA currently does not perform inter-procedural VSA to track stack pointers that cross the function boundary. Effectively, those pointers are treated as pointers to the heap-global region. This approximation is sufficient for MazeRunner's attack surface calculation as its attack models assume that the heap-global region can be manipulated arbitrarily. Treating a stack pointer as if it were a pointer to the heap-global region gives the attacker more power and thus overapproximates the attack surface.

Transforming to a stack-free CFG. To get a stack-free CFG, we first introduce new variables to a function that represent the function's alocs in its stack frame. MazeRunner's naming convention is to use name $f_{[o1,o2]}$ for a new variable that is introduced for the aloc that has offset range $[o1, o2]$ within function f.

For any memory access in the function, we use PSA's result to determine whether it is a stack access. If it is a stack memory access, PSA's result is used

to determine what stack alocs are accessed and how to convert the stack access into variable accesses. If only a single aloc is accessed, then the memory access is converted to a deterministic access to the corresponding new variable that was introduced. As an example, suppose the program has a 4-byte memory-write instruction "Mem[EBP − 4] = \$2" in function `f` and EBP is determined by PSA to have a single offset -4. Further assume that there is a 4-byte aloc at stack offset range [-8,-5] and for that aloc we introduce a new variable $\mathtt{f}_{[-8,-5]}$. Then the stack access "Mem[EBP − 4] = \$2" is converted to $\mathtt{f}_{[-8,-5]} = \$2$, a variable access.

Although rare, it is possible that PSA might say that a memory access can possibly access different alocs (on different control-flow paths). For instance, PSA might say that, before "Mem[EBP − 4] = \$2", EBP can have either offset -4 or -8. So the memory access can touch either the aloc at [-8,-5] or the one at [-4,-1]. In this case, we convert the memory write into a nondeterministic operation that accesses either the variable for aloc [-8,-5] or the variable for aloc [-4,-1]: $(\mathtt{f}_{[-8,-5]}|\mathtt{f}_{[-4,-1]}) = \$2$. This overapproximates the behavior of the original program.

4.3.4 Attack-aware dependency tracking

MazeRunner's dependency tracking decides, at every program point, what storage locations (registers and stack slots) have dependency on the policy, meaning that values in them may affect whether the security violation can happen at the target. Since the input is an RTL-level stack-free CFG, both registers and stack slots are encoded as RTL locations in the CFG. ADT also takes the attack instructions into account when determining what is dependent. Since these attack instructions were inserted according to the attack model, the dependency tracking is attack aware.

Initially, at the target instruction, the storage locations mentioned in the attack condition are in the dependency set. As an example, suppose the attack condition is "eax & $0x6 = 0x6$". Then eax is in the initial dependency set. Based on the initial dependency, the tracking goes backward from the target instruction in the CFG and propagates dependency. For RTL instructions, the dependency tracking rules follow data flow and are essentially backward taint tracking. As a simple example, if the instruction is "eax = ebx" and eax is tainted after the instruction, then ebx becomes tainted before the instruction, meaning that ebx's value before the instruction may affect the final policy outcome.

The operational semantic of attack instructions is to modify the memory according the specified attack model, which introduces attacker's influence into the program. During dependency tracking, attack instructions have the effect of cutting off the program's original dependency and adding the influence from the attacker. Tracking rules for attack instructions are designed to reflect attacker's influence on the program. This can be illustrated via an example:

$$x = \text{eax}; \text{sATK}_{\text{all}};$$
$$// \text{ assume x is in the dependency set here}$$

Further assume that x is a variable that represents a stack slot (i.e, introduced during the conversion to the stack-free CFG). The instruction $x = \text{eax}$ stores the value of eax into x, meaning that x should depend on eax. However, the stack-attack instruction "sATK_{all}" can modify x right away and as a result the effect of $x = \text{eax}$ is canceled by the stack attack. Therefore, x no longer depends on eax after the attack. Suppose x is in the dependency set after the two instructions. When processing sATK_{all}, ADT would determine that the dependency set before the instruction should not contain x but contains a special mark representing the attacker's influence ("ATK"); when processing $x = \text{eax}$, since ATK (not x) is in the dependency set, eax is not added to the dependency set at the point before $x = \text{eax}$. This way, the dependency between x and eax is cut off and the attacker's influence is reflected.

In general, when processing sATK_{α} with a post dependency set ϕ, the dependency set beforehand is the result of replacing in ϕ the stack slots specified by α with ATK. Another source of ATK is memory updates on the heap-global region. Since the attack model (memwrite,all) inserts a hgATK instruction after a memory update, the memory update immediately loses its effect after hgATK. So, the dependency tracking treats such memory updates as no-ops. Then, since the heap-global region is controlled by the attacker, memory reads introduce "ATK". Together with the fact that the stack is abstracted away, the ADT does not need to model memory. Not modeling memory is the key to scalability but unavoidably introduces imprecision. However, we will show that the remaining precision is sufficient for the goal of comparing different CFI designs.

After ADT is finished, we use the dependency sets to classify program points. If the dependency set at a program point contains the attacker mark ("ATK"), we

classify it as risky; otherwise, it is treated as safe.

Formal rules. A set of formal tracking rules are presented in Table 4.1. Notation "adt(i, Q_1) = Q_2" means that if the dependency set *after* instruction i is Q_1, then the dependency set *before* i should be Q_2. The tracking rule for an assignment to RTL locations $loc = e$ replaces the destination location (loc) in the dependency set with the variables in the source expression (Var(e)). This rule also applies to stack slots introduced by our stack-free conversion. As for the memory updates Mem[e_1] = e_2, since the attack model (memwrite, all) inserts an hgATK instruction after a memory update, the memory update immediately loses its effect after hgATK. Thus, the rule for Mem[e_1] = e_2 followed by hgATK treats the two instructions together as a no-op. The rule for the IF-DO instruction IF e DO i introduces variables in the condition expression, Var(e), into the dependency set and recursively applies ADT to the instruction i. The rule for the stack attack instruction replaces the affected variables in the dependency set with a special mark "$atkmark$", which represents the attacker's influence. Here, affected variables (affectedVars(α, `f`)) are stack slots related to the function `f` specified by α.

The rules for RTL expressions determine what variables should be involved. Variables are constant bit-vectors, RTL locations, and stack slots; they are retrieved from load_loc(loc) and bitvec(sz, v) expressions. As for the arithmetic expression (arith($bvop$, $e1$, $e2$)), the Boolean test expression (test(cmp, $e1$, $e2$)), and the if-then-else expression (ite($cond$, e_1, e_2)), their rules recursively apply the variable retrieval process to their sub-expressions. The case for memory read is more interesting. Since all memory updates are treated as no-ops and the global-heap region is controlled by the attacker, any memory read can be arbitrary, which generates the "ATK" mark.

To deal with possible loops in a CFG, we compute a fixpoint regarding to dependency sets. Since the amount of storage locations in a CFG is finite, the space of dependency sets forms a finite lattice; this guarantees termination of the fixpoint computation.

Optimizations. To further accelerate the ADT, we employed a series of optimizations. First, if the dependency goes to hard-encoded data, we do not track such data in the dependency set. As a result, a dependency set may become empty and the tracking for that branch can terminate early. For example, if the dependency

Dependency tracking for instructions:
$\text{adt}(loc = e,\ Q) = Q[\text{Var}(e)/loc]$
$\text{adt}(\text{Mem}[e_1] = e_2; \text{hgATK},\ Q) = Q$
$\text{adt}(\text{IF } e \text{ DO } i,\ Q) = \text{Var}(e) \cup \text{adt}(i,\ Q)$
$\text{adt}(\text{sATK}_\alpha,\ Q) = Q[\text{ATK}/\text{affectedVars}(\alpha, \texttt{f})]$
where ATK is a mark for attacker's influence,
and `f` is the function being analyzed
Retrieve variables from RTL expressions:
$\text{Var}(\text{arith}(bvop,\ e1,\ e2)) = \text{Var}(e1) \cup \text{Var}(e2)$
$\text{Var}(\text{test}(cmp,\ e1,\ e2)) = \text{Var}(e1) \cup \text{Var}(e2)$
$\text{Var}(\text{load\_loc}(loc)) = \{loc\}$
$\text{Var}(\text{load\_mem}(addr)) = \text{ATK}$
$\text{Var}(\text{ite}(cond,\ e_1,\ e_2)) = \text{Var}(cond) \cup \text{Var}(e_1) \cup \text{Var}(e_2)$
$\text{Var}(\text{bitvec}(sz,\ v)) =$ constant bit-vector v of size sz

Table 4.1. Attack-aware dependency tracking rules.

set after "eax = 2" is {eax}, then ADT determines that the dependency set before should be empty (as only constants can affect the value of eax).

Second, if the dependency set contains the attacker mark, the whole set can be simplified into a singleton set with the mark. This is an optimization for our design choice of program-point classification. Since the attacker mark can never be cleared, all predecessor program points will have the mark in their dependency set. According to our classification principle, having the attacker mark for a program point means risky. So, all predecessor program points will be risky. In this case, our dependency tracking can then be simplified to be a backward reachability analysis on the CFG, which is more efficient.

Third, when contexts switch between different functions, we employ a filter process to avoid tracking storage locations that are only related to the current context. This optimization can avoid tracking storage locations in an unrelated context. For example, when the dependency tracking backwardly follows a return edge, if eax is not in the dependency set before entering the new context, the dependency set for the new context will be empty. Our tracking can safely skip the call instruction and stay within the current context. This optimization is also tailored for our setup by assuming a calling convention and because we assume attack models that can control global regions.

4.4 Security Metric Design

A precise attack-surface metric should consider the number of risky program points as well as paths through which attacks might happen. However, the number of paths from a risky program point to the destination may be infinite; even if it is not, counting all paths in large programs is not scalable. Therefore, we propose to limit the length of paths that are used for our metric computation. We call this metric, k-step attack surface:

Definition 1 (k-step attack surface) *A k-path in a CFG is defined to be a path of k edges. We write $Path(k, V, E)$ to be the set of k-paths in the CFG ($G = (V, E)$). Further, a risky basic block is defined to be one that contains at least one risky program point. We write $RB(V, E, A, C)$ for the set of all risky basic blocks discovered by our ADT in the CFG under the attack model A, with the policy C.*

If a k-path connects two risky basic blocks, it is defined to be an attack-facilitating k-path; we define the k-step attack surface for a CFG under an attack model A against a policy C, written as $AS(k, V, E, A, C)$, as the set of all attack-facilitating k-paths. It can be formalized as $AS(k, V, E, A, C) = \{(b_1, b_2, \ldots, b_{k+1}) \in Path(k, V, E) \mid b_1, b_{k+1} \in RB(V, E, A, C)\}$. Finally, our attack-surface metric is the number of paths in $AS(k, V, E, A, C)$.

Essentially, this metric measures the amount of freedom when going from one risky basic block to the next risky basic block, if k steps are allowed in the CFG. Since ADT is conservative, basic blocks that are not risky are free from attacks towards the given security-violation policy. Thus, attack paths can pass only risky basic blocks. When $k = 0$, the metric becomes the number of risky basic blocks, since a basic block can reach itself in zero steps. When $k = 1$, the metric estimates the freedom of chaining control-flow edges at every step during attack. We can imagine that an attacker is using some greedy strategy to search for attack paths. Starting from a risky basic block, the attacker can choose only successor basic blocks that are also risky. Thus, the 1-step attack surface estimates the attack surface for such an attacker. When $k > 1$, the attack-facilitating k-paths can be treated as chains of risky basic blocks.

4.5 Limitations and Discussions

This work is not a perfect solution to the challenging but important topic of CFI policy security evaluation. Thus, MazeRunner does have several limitations, which are discussed here.

Attack model granularity. A more fine-grained labeling system of attack models might help a more precise evaluation of CFI defenses. For instance, for the stack region we can add a top label that allows the attacker to modify only the top frame on the stack. As another example, we can increase the granularity of memory separation. For instance, we can further separate the heap region from the global data region or use different `malloc` sites to partition the heap memory region. However, supporting fine-grained attack models may require analyzing shared libraries in addition to the application code.

Security violation comprehensiveness. We believe considering all possible attacks is impractical for security evaluation. In this dissertation, we aim to demonstrate the advantages of MazeRunner's approach with 4 representative policies. However, to perform a customized or more thorough security evaluation, more policies would be needed. For example, to evaluate the attack surface of sensitive information leakage, one may specify a policy about `read` and `write` functions to capture the critical points of information flow.

Defense modeling and program analysis accuracy. More advanced control-flow defenses are not modeled in MazeRunner, such as path-sensitive CFI (e.g., [50]) and memory-protection techniques for protecting control flow (e.g., CPI [53]). We believe that our attack-surface metric is still applicable when considering those defenses by designing new attack models with more accurate attack-aware program analysis. For example, though we have shown that the attack-aware data dependency tracking helps improve the evaluation precision for statically determined CFI policies, the analysis is path-insensitive so that it is not sufficient for evaluating path-sensitive defenses. To enable path sensitivity, for instance, employing symbolic execution or a customized value-set analysis might be the key.

Metric Design. Though our way of synthesizing CFG information with the risky program points into one metric is demonstrated to be meaningful, there are

opportunities to improve the metric design. For example, one possibility is to give weights to different risky program points to represent their different degrees of severity. The principle for assigning weights can rely on expert knowledge or program analysis, which can be another research problem for security evaluation.

Application scope. We have not tested MazeRunner with binaries compiled from C++ programs due to the limitation of the chosen CFG construction tool [15]. However, the key technique, attack-aware data dependency tracking, is designed on an intermediate language (RTL), which is independent from the source code. Therefore, we believe extending MazeRunner to support C++ binaries only requires changes in the stack-free transformation. In addition, CFG construction for C++ binaries is necessary, which, however, is an orthogonal topic. On the other hand, MazeRunner has no support for x86-64 binaries, because there was no support of translating x86-64 instructions into RTL instructions in prior work [1].

4.6 Summary

To push CFI evaluation forward, we propose MazeRunner, a framework for quantitatively evaluating the attack surface of a CFI-protected program. In contrast to traditional metrics, our metric provides an overapproximated estimation and considers how gadgets can be chained to form an attack path. We propose a novel attack-aware dependency tracking for a fine-grained attack-surface evaluation, in which attacker's influences are considered. Since our attack models are relatively stronger than real-world attackers and our system is designed to overapproximate attack surface, our metric is conservative but meaningful for measuring the insecurity of a program.

Compared to the original paper published in TrustCom 2021 [34], this chapter contains more technical details and the evaluation section is moved to Chapter 6.

Chapter 5
Risky Paths as Attack Surface

5.1 Overview

In this chapter, we propose another metric that is computed based on a per-path value tracking analysis (PVTA). For this metric, we define the attack surface as all risky paths, which is a more fine-grained way than using risky program points to represent the attack surface. The intuition is that one risky program point may correspond to multiple risky paths and different risky program points may be mapped to different numbers of risky paths. Moreover, the attack model assumed for this metric has less power than what MazeRunner assumes, in the aspect of when the memory corruption can happen. In terms of the accuracy of the underlying analysis, compared to the attack-aware dependency tracking analysis discussed in Section 4.3, PVTA is context-sensitive and partially path-sensitive. In all, the new metric is pursuing a more precise interpretation of the attack surface than what MazeRunner does.

5.1.1 Threat Model

For this metric, we target at low-level attacks that are triggered by memory corruptions in a victim program, such as buffer overflow, format string manipulation, and use-after-free vulnerabilities. We split such an attack into two phases: memory manipulation and attack launch. In the first phase, the attacker exploits the memory corruption vulnerability (may repeatedly execute the vulnerability code)

to manipulate the runtime memory into a shape desired by the attacker. Then, in the second phase, the attacker launches the attack by letting the program execute without extra interference to achieve a malicious goal, such as controlling an indirect branch's target, propagating the arbitrary memory write capability, and controlling an argument of a sensitive library call. Thus, we define the following threat model to reflect the aforementioned attack scenario.

Memory-Corruption Phase. We assume that the victim program contains a memory-corruption vulnerability, which can be exploited by the attacker to gain an arbitrary write primitive. The arbitrary write primitive allows the attacker to set up the runtime memory in an arbitrary way, on top of which, all registers are also deemed as manipulable by the primitive except the stack pointer. Such a primitive is more powerful than the arbitrary memory write primitive commonly assumed by prior work [2–10]. For example, BOPC [9] assumes the attacker can perform an arbitrary memory write at any time but only once to discover chains of basic blocks that achieve a user-specified functionality. However, we believe our assumption is still reasonable, because it is consistent with the memory-corruption phase. With the execution of instructions that transfer data between memory and registers, it is possible that registers are also configured with attacker-desired values. For example, when a stack-based buffer overflow happens, the stack slot that stores the previous stack frame's base address can be overridden and can be later popped to the RBP register in an x64 machine after the current function returns. In all, we use the arbitrary write primitive to summarize the attacker's influence to the victim program during the memory-corruption phase.

Attack-Launch Phase. To simulate the second phase, we further constrain that the attacker can make no influence to the program's execution after the arbitrary write primitive. Therefore, the attacker's goal can only be achieved by finding a legitimate path that starts from the arbitrary write primitive with the crafted memory and ends at a target program point where the program state triggers the intended security violation, e.g., an indirect call instruction with the operand holding a value sourced from the crafted memory. Without CFI, there exists a gigantic number of legitimate paths that can achieve the attacker's goal, since indirect branches can target any executable code address. In contrast, in this work, we assume a context-sensitive CFI implemented by a shadow stack or hardware

support is enforced to protect the victim program, which reduces a large number of illegitimate paths. Therefore, during the second phase, the attacker cannot rely on paths that violate the enforced CFI, which increases the difficulty for the attacker to accomplish the malicious goal.

5.1.2 System Overview

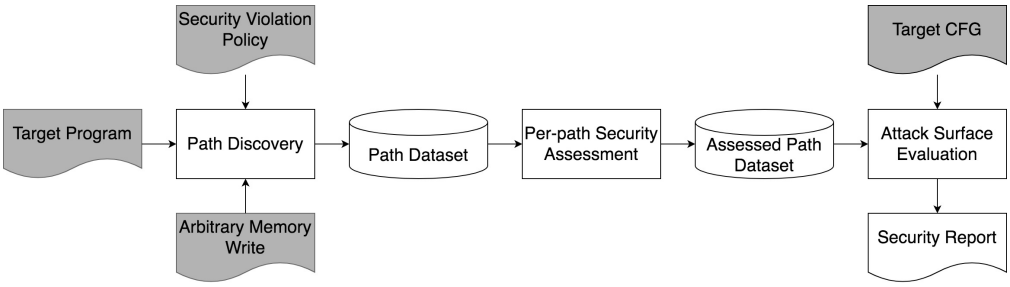


Figure 5.1. System Overview

We call the system that computes the metric, SpaceExplorer, which has three components: path discovery, per-path security assessment, and attack-surface evaluation. Figure 5.1 shows a workflow of this system. The path-discovery component first generates for the target program a base CFG where targets of indirect branches are resolved based on relocation information (i.e., the Address-Taken CFG supported by Section 3.6). Then, it discovers connecting paths that start from program points where the memory becomes arbitrary due to arbitrary write exploits and stop at program points where a specified security violation may happen. In the second component, SpaceExplorer performs a per-path value tracking analysis (PVTA) to assess if a connecting path is safe or risky. Risky paths consist of a superset of ground-truth attack paths among the discovered connecting paths; in other words, PVTA overapproximates attack paths and achieves a 100% precision in determining safe paths among the discovered connecting paths. Then, the number of risky paths is used to measure the baseline attack surface of a program; that is, we assume that the base CFG exposes the largest attack surface. Note that any sound CFG should be a subset of the base CFG. Therefore, in the final component, given a target CFG, the system computes the number of prevented risky paths to measure the remaining attack surface. Next, we explain the specifications of four inputs and the final output. The technical details of the

three components are left to Section 5.2. Section 5.3, and Section 5.4.

The *target program* is fed into SpaceExplorer as binary code. SpaceExplorer disassembles the program and constructs a base CFG for it. The base CFG construction is designed as a parameterized module. For example, the base CFG can be generated either by analyzing the binary code itself [12, 18] or the relocation information [32, 67], depending on the tool embedded. In our implementation, we use the the base CFG precision provided by our flexible CFG construction work [67] (discussed in Chapter 3) to disassemble the binary code into a base CFG, which is represented by a set of basic blocks and edges between basic blocks.

Each basic block holds a sequence of assembly instructions with their addresses and operational semantics specified in RTL (register transfer list) language [1] (details are in Section 4.3.1). Each basic block is also attributed with its control-flow successors. Since the targets of indirect calls and indirect jumps are retrieved from relocation information and return targets are resolved by constructing call graphs, the base CFG should cover all legitimate control-flow edges; that is, the edge set of any CFG valid for CFI is a subset the base CFG's edges. Hence, a *target CFG* is represented by a subset of control-flow edges in the base CFG; and each edge is a pair of basic blocks of the base CFG.

The *arbitrary write primitives* are program points specified by the users, either based on true memory-corruption vulnerabilities or at the users' discretion. In our implementation of SpaceExplorer, we allow two methods of providing input arbitrary write primitives. One way is to provide arbitrary write primitives through an user interface in terms of the code addresses; the other way is to let SpaceExplorer randomly select a percentage of basic blocks as arbitrary write primitives for a general attack-surface evaluation. As discussed in Section 5.1.1, an arbitrary write primitive marks the end of a successful memory-corruption phase.

The *Security-violation policies* can be categorized into two genres: general policies and concrete policies, which is similar to what we have done for MazeRunner. General policies are defined for a type of assembly instructions; controlling the operands of some assembly instructions can bring the attacker the convenience in launching a complicated attack. For example, we can define a general policy for indirect call instructions and the attacker's goal is to connect an arbitrary write primitive to one of the indirect call instructions, while making sure that the value of the indirect-call operand is controllable by the attacker. In contrast, concrete

policies are designed for concrete system calls, such as `mprotect` and `execve`; being able to manipulate their arguments can cause great damage to the security of the victim program. For example, we can define a concrete policy for `mprotect` calls, in which the goal is to set the third argument (arg_3) to satisfy $arg_3 \ \& \ 0x6 \ = \ 0x6$. Such a behavior can make a memory region both writable and executable for the attacker to inject code.

5.2 Path Discovery

The goal of this component is to discover static paths that connect an arbitrary write primitive to a security-violation instruction in a base CFG. Each connecting path is represented by a sequence of basic blocks. The connecting path discovery problem is similar to a classical path finding problem in a directed graph. However, our threat model complicates the problem by constraining that the callsites and return targets should match in a path. To do so, we maintain a stack to record return addresses to make sure every return instruction only goes back to its most recent caller during the static path discovery, which simulates context-sensitive CFI.

Though such a context-sensitive path finding problem is intuitive, an efficient implementation is not trivial. First, traditional shortest path algorithms, such as the Dijkstra algorithm, are not suitable for this task, since such algorithms do not consider the context-sensitivity. On the other hand, due to the notorious path explosion problem, a basic graph traversal implementation would easily exhaust runtime memory for large programs. To bypass such an obstacle, we resort to performing random walks to explore the graph for connecting paths. In detail, our algorithm starts from each security violation point and backwardly follows control-flow edges to construct context-sensitive paths that connect the security violation point to one arbitrary write primitive. At each node that contains multiple outgoing edges, the algorithm randomly selects one to follow. Moreover, the algorithm avoids trapping into a loop; in the implementation, we have a parameter to control the maximum allowed loop iterations.

This method has its own disadvantages: 1) it does not guarantee to find all possible paths; 2) it may discover repeated paths; and 3) path conditions are not validated. To overcome such disadvantages, we let this component run as long as

possible to increase the completeness of path discovery. However, we cannot let the system get stuck in component one. So, for a certain amount of attempts of random walks, we record all the discovered connecting paths into a dataset and start the rest of components. When the second and third components finish, we restart the path-discovery component and try to add more data entries to the dataset. The number of iterations can be decided by the user or by setting up some stopping criteria. We also store the hash of each path along with the path in the dataset so that the system can avoid adding repeated paths to the dataset. We leave the path condition checking to the per-path value tracking component, which has two reasons. First, random walks are likely to produce repeated paths, which would incur repeated path condition checking for the same path. Second, checking path conditions requires tracking program states, which is overlapped with the per-path value tracking component. Thus, separating path condition checking out of path discovery can better modularize the two components. Because of our design choice, we can foresee two improvements of this component. First, we can implement the system in a pipeline structure so that different components can work concurrently. Second, to improve the efficiency of the path discovery, we could also employ reinforcement learning to be more focused on paths that are likely to be risky.

Since there is no guarantee for discovering all connecting paths in a program, the paths that our metric relies on may be incomplete. However, due to the randomness, with enough sampling rounds, the discovered connecting paths should be representative for the whole sample space. In evaluation, we use the code coverage to give a basic sense of completeness.

5.3 Per-path Security Assessment

For this component, the input is a path consisting of basic blocks; and each basic block contains a list of RTL instructions. PVTA (per-path value tracking analysis) evaluates each RTL instruction with a pre-state and calculates a post-state. Thus, with an initial state, PVTA can infer the final state of a path. Based on the security-violation policy and the final state, SpaceExplorer checks if the security-violation state is achievable at the end of the path. If it is not achievable, SpaceExplorer determines the path to be safe; if it is achievable, the path is deemed as potentially

$$
\begin{array}{rcl}
v & \in & \text{Integers} \\
sz & \in & \text{Integers} \\
ndx & \in & \text{Integers} \\
reg & := & \text{EAX} \mid \text{EBX} \mid \ldots \\
flag & := & \text{CF} \mid \text{ZF} \mid \ldots \\
mloc & := & \text{S}[v] \mid \text{G}[v] \mid \text{H}_{ndx}[v] \\
loc & := & \text{PC} \mid reg \mid flag \\
st & := & loc \mid mloc \\
val & := & bv(sz, v) \mid \&mloc \mid \top
\end{array}
$$

Figure 5.2. The storage locations and values considered in a program state.

unsafe; if SpaceExplorer cannot make a clear decision, the path is conservatively treated as a suspicious path. Potentially unsafe and suspicious paths are classified as risky paths. At a high level, PVTA is an overapproximated substitution of symbolic execution.

Program State Modeling. We represent a state with a mapping from storage locations to values. SpaceExplorer tracks four kinds of storage locations: registers/flags, stack memory locations, data-section memory locations, and heap memory locations. Values have three categories: bitvector, address of a memory location, and $\top$ (top), where $\top$ represents an arbitrary value. The representation of storage locations and values in a program state is formalized in Figure 5.2.

A register/flag is represented by its name. At the binary level, there are three types of memory accesses: stack accesses through stack pointers, data-section accesses through concrete memory addresses, and heap accesses through heap pointers. We assume the three kinds of memory accesses do not overlap. Thus, we can partition the memory into multiple memory regions, each of which is modeled as an array. The stack is represented by an array S[−]; and a stack memory location (i.e., a local variable) is represented by an indexed slot of the stack array. For example, S[100] represents the stack memory location that is 100 bytes above the top address of the stack at the beginning of the path being evaluated. We use another array, G[−], to represent memory locations in the global data sections. For example, G[1000] is the memory location at address 1000. We call dynamically allocated memory regions by `malloc`-family functions as heap regions and represent them with arrays. Heap memory locations are represented by indexed slots of these arrays. Since in a path there could be multiple heap regions allocated, we use

indices to distinguish different heap regions. Indices are generated during evaluating the malloc-family functions in a path. `malloc` and `calloc` increment the index but `realloc` does not. For example, $H_0[100]$ is a heap memory location with offset 100 in the heap region allocated by the first `malloc` or `calloc` in a path, while $H_1[20]$ is a heap memory location with offset 20 in the heap region allocated by the second `malloc` or `calloc` in the path. PVTA can determine what memory locations are accessed by RTL instructions based on the program state. Details will be provided in the discussion of value tracking rules. Note that PVTA's memory modeling does not change the representation of an input path. The arrays are how memory locations are modeled in a program state. Compared to the one-array memory modeling used in classic symbolic execution, our region-based modeling does not need to add logic constraint to distinguish stack pointers and heap pointers, or to concretize these pointers based on heuristics.

A bitvector value represents a numerical value; in implementation, we use an unsigned integer with a size to represent it. An address of a memory location is represented by a & mark along with a memory location; this notation is similar to C's syntax of taking the address of a variable. Note that a memory location is similar to a one-byte variable. For example, &S[100] represents the memory address of the stack location at offset 100 to the top address of the initial stack. $\top$ appears when PVTA cannot decide the actual value. For example, a memory read from a memory location controlled by the attacker would yield a $\top$.

Since every input path starts with an arbitrary write primitive, according to our threat model, the initial program state should be that ESP is mapped to the top address of the current stack, represented by &S[0], with other storage locations initialized to be $\top$. However, at the beginning of a path, we do not know what storage locations will occur in a path. Thus, adding concrete mappings for these storage locations is impossible. Our solution is to treat a non-tracked storage location as being mapped to $\top$. As a result, the initial program state is simply that ESP is mapped to &S[0].

Value Tracking Rules. In Table 5.1, we present the evaluation rules of PVTA. RTL instructions may modify the program state, while expressions evaluate to values given a program state. We use pvt(Γ, *ins*) to represent the rule of evaluating an instruction and Val(Γ, exp) for evaluating an expression, where Γ represents a

Value tracking for RTL instructions (pvt(−, −)):

pvt(Γ, $loc = e$) := $\Gamma[loc \rightarrow \text{Val}(\Gamma, e)]$
pvt(Γ, Mem[e_{addr}] = e) :=
    if Val(Γ, e_{addr}) == $bv(sz, v)$ then $\Gamma[\text{G}[v] \rightarrow \text{Val}(\Gamma, e)]$
    elif Val(Γ, e_{addr}) == $\&mloc$ then $\Gamma[mloc \rightarrow \text{Val}(\Gamma, e)]$
    else $\forall\ mloc,\ \Gamma[mloc \rightarrow \top]$
pvt(Γ, IF e DO i) :=
    if Val(Γ, e) == $\top$ then
        randomly pick one from pvt(Γ, i) and Γ
    elif Val(Γ, e) == 1 then pvt(Γ, i)
    else Γ

Value of RTL expressions (Val(−, −)):

Val(Γ, arith($bvop,\ e_1,\ e_2$)) :=
    if Val(Γ, e_1) == $bv(sz_1, v_1)$ and Val(Γ, e_2) == $bv(sz_2, v_2)$ then
        BitArith($bvop, bv(sz_1, v_1), bv(sz_2, v_2)$)
    elif Val(Γ, e_1) == $bv(sz_1, v_1)$ and Val(Γ, e_2) == $\&arr[v_2]$ then
        let $bv(sz', v')$ = BitArith($bvop, bv(sz_1, v_1), bv(32, v_2)$) in
        $\&arr[v']$
    elif Val(Γ, e_1) == $\&arr[v_1]$ and Val(Γ, e_2) == $bv(sz_2, v_2)$ then
        let $bv(sz', v')$ = BitArith($bvop, bv(32, v_1), bv(sz_2, v_2)$) in
        $\&arr[v']$
    else $\top$
Val(Γ, test($cmp,\ e_1,\ e_2$)) :=
    if Val(Γ, e_1) == $bv(sz_1, v_1)$ and Val(Γ, e_2) == $bv(sz_2, v_2)$ then
        BitArith($cmp, bv(sz_1, v_1), bv(sz_2, v_2)$)
    elif Val(Γ, e_1) == $\&arr[v_1]$ and Val(Γ, e_2) == $\&arr'[v_2]$ then
        if $arr == arr'$ && $v_1 == v_2$ then $bv(1, 1)$ else $bv(1, 0)$
    else $\top$
Val(Γ, load_loc(loc)) := $\Gamma[loc]$
Val(Γ, load_mem(e_{addr})) :=
    if Val(Γ, e_{addr}) == $bv(sz, v)$ then $\Gamma[\text{G}[v]]$
    elif Val(Γ, e_{addr}) == $\&mloc$ then $\Gamma[mloc]$
    else $\top$
Val(Γ, ite($e_{cond},\ e_1,\ e_2$)) :=
    if Val(Γ, e_{cond}) == $\top$ then
        randomly pick one from Val(Γ, e_1) and Val(Γ, e_2)
    elif Val(Γ, e_{cond}) == $bv(1, 1)$ then Val(Γ, e_1)
    else Val(Γ, e_2)
Val($\Gamma, bv(sz, v)$) := $bv(sz, v)$

Table 5.1. Rules of the Per-path Value Tracking Analysis

program state. The syntax of RTL can be found in Figure 4.5.

The rule of an assignment-to-location instruction, $loc = e$, updates the program state with the storage location loc being mapped to a new value produced by e, i.e., $\text{Val}(\Gamma, e)$. We only consider the program counter (PC), registers, and flags in a program state. The rule of a memory write instruction, $\text{Mem}[e_{addr}] = e$, has four cases: if the memory-address expression is evaluated to a bitvector, i.e., $\text{Val}(\Gamma, e_{addr}) == bv(sz, v)$, the program state is updated with a data-section memory location, $\text{G}[v]$, being mapped to $\text{Val}(\Gamma, e)$; if the memory address expression is evaluated to an address of a memory storage location, i.e., $\text{Val}(\Gamma, e_{addr}) == \&mloc$, in the new program state, a memory location, $mloc$, is mapped to $\text{Val}(\Gamma, e)$; otherwise, the address expression is overapproximated to a $\top$ and we map all memory locations to $\top$. Intuitively, this rule first determines what memory location (i.e., either a stack location $\text{S}[n]$, a data-section location $\text{G}[n]$, or a heap location $\text{H}_i[n]$) is accessed by a memory write and then updates the program state accordingly. The determination is to evaluate the address expression based on the program state. As for the conditional instruction, IF e DO i, there are also three cases. First, if the condition expression is evaluated to a $\top$, we randomly decide whether to further evaluate the sub-instruction i, unless the sub-instruction i is setting the program counter; instead, PVTA takes the Γ that would make the path condition satisfied. Second, if the condition expression evaluates to a bitvector of value 1, we further evaluate instruction i; otherwise, the program state stays unchanged. Note that in implementation PVTA removes a record from the program state if a storage location is mapped to $\top$.

Evaluating an arithmetic expression $\text{arith}(bvop,\ e_1,\ e_2)$ has four cases. First, if the two operands both evaluate to bit-vectors, the evaluation of the arithmetic expression is to compute the result of the bit-vector operation specified by $bvop$ on the two bit-vectors. In the second case, if e_1 evaluates to a bit-vector $bv(sz_1, v_1)$ and e_2 evaluates to an address of a memory location $\&arr[v_2]$, where arr represents one of the three kinds of arrays for modeling memory regions, the result of evaluation is an address of a new memory location $\&arr[v']$, where the new index v' is the result of the bit-vector operation $bvop$ on the bit-vector $bv(sz_1, v_1)$ and the index of the memory location v_2. The third case is similar to the second case with the only difference being the order of the bit-vector and the memory-location address. For all other situations, the evaluation result is over-approximated to a $\top$. Evaluating

a comparison expression test($cmp,\ e_1,\ e_2$) has three cases. When the two operands both evaluate to bit-vectors, the evaluation is to perform a comparison operation *cmp* on the two bit-vectors. When the two operands both evaluate to addresses of memory locations, the evaluation result is a bit-vector 1 if and only if the two memory locations are the same. Otherwise, the result is a $\top$. A location-load expression load_loc(loc) simply evaluates to the mapped value of *loc* in the program state Γ. The evaluation of a memory-load expression load_mem($addr$) has three cases. If the address expression evaluates to a bitvector $bv(sz, v)$, the output value is the mapped value of $\mathrm{G}[v]$ in Γ. If the address expression evaluates to an address of a memory location $\&mloc$, the output value is the mapped value of *mloc* in Γ. Otherwise, the output value may be anything, i.e., $\top$. There are three cases for the rule of conditional (if-then-else) expression ite($e_{cond},\ e_1,\ e_2$). If the condition expression is evaluated to a $\top$, the output value is randomly selected from the evaluated values of the two operand expressions. If the condition expression evaluates to a bitvector of value 1, the first operand is evaluated as the final output value of the whole expression. Otherwise, the second operand is evaluated as the output. For a bitvector expression, it simply evaluates to the bitvector, which is represented by an unsigned integer with a size.

Library Function Modeling. Calls to `malloc`-family functions have special rules. The invocation of `malloc` or `calloc` function returns a $\&\mathrm{H}_i[0]$ to the EAX register, i.e., $\Gamma[\mathrm{EAX} \rightarrow \mathrm{H}_i[0]]$, and increments the counter i. In contrast, `realloc` returns the first argument to EAX, i.e., $\Gamma[\mathrm{EAX} \rightarrow arg_1]$, where arg_1 represents the storage location of the first argument of the `realloc` callsite. Function calls to other shared libraries are modeled as special instructions; we call them lib-call instructions. The operational semantics of lib-call instructions are based on whether the library function returns a value or modifies the memory. Given different answers on the two dimensions, we have four lib-call instructions: value-return memory-modified, value-return memory-unchanged, void-return memory-modified, and void-return memory-unchanged. A value-return memory-modified instruction updates the program state Γ by mapping the storage location that holds the return value and all memory storage locations (i.e., the three kinds of arrays) to $\top$, thus being removed from the program state; for example, x86 calling convention assumes EAX for returning a value. For example, if a library function takes a pointer parameter

and modifies the memory through the pointer, such as `memset`, it is modeled as a value-return memory-modified instruction, which overapproximates the effect of memory modification of the library function. A value-return memory-unchanged instruction only removes the storage location of return value; a void-return memory-modified instruction only removes memory storage locations; and a void-return memory-unchanged instruction is simply a no-op. In implementation, we classify each library function call on demand into one of the lib-call instructions according to the specification of the target library function.

Path Condition Checking. PVTA evaluates static paths sampled in the path discovery component, where path conditions are not considered during the random walks for efficiency. A sampled path is a sequence of connected edges in a CFG. However, two connected edges may contradict each other due to the path conditions. For example, the then-branch of a conditional jump with a condition as $x < 0$ is contradicting to the then-branch of another conditional jump with a condition as $x > 10$ if the variable x is not changed between the two conditional jumps. Even if there are 2×2 static paths that pass the two conditional jumps in the CFG, there is one infeasible path. Thus, PVTA needs to check path conditions. Path conditions determine what branches to take for conditional jumps, indirect jumps, and indirect calls. During runtime, if a condition is satisfied, the program counter is set with the correct code address. PVTA makes use of this observation to check path conditions. In detail, at each basic block, PVTA checks if the code address of the current basic block matches the program counter (PC) tracked in the current Γ. If they do not match, the path under assessment is not feasible thus safe. Otherwise, either the two values match or the tracked PC is a $\top$ (may happen after an indirect call), indicating that the path is so far valid.

For example, suppose we have two consecutive conditional jumps as follows.

```
0: x = 10
1: if x > 0 goto 3
2: y = 1
3: if x < 10 goto 5
4: y = 2
5: y = 3
```

There are four paths from instruction 0 to 5 in the example: [0, 1, 2, 3, 4, 5], [0, 1,

2, 3, 5], [0, 1, 3, 4, 5], and [0, 1, 3, 5]. Since at instruction 0, x is assigned with 10, among the 4 paths, only [0, 1, 3, 4, 5] is valid. We use path [0, 1, 3, 5] to show how PVTA determines it to be infeasible. At the binary level, "goto i" is to set the program counter (PC) to be i. Thus, PC is treated as a storage location and tracked in the program state. To evaluate path [0, 1, 3, 5], the program state is initialized to be $\{\text{PC} : 0\}$. The first instruction to be evaluated in the path is at address 0, which matches PC's value in the initial program state; thus, instruction 0 is allowed to be evaluated. After evaluating instruction 0, the program state becomes $\{\text{PC} : 1; \text{x} : 10\}$. Next, PVTA needs to evaluate the second instruction in the path, which is at address 1. We can see that PC's value in the current program state matches the address of the target instruction. Thus, according to the current program state, the condition of instruction 1 is evaluated to be `true` and the "goto 3" is evaluated. As a result, the program state becomes $\{\text{PC} : 3; \text{x} : 10\}$. The next instruction in the path to be evaluated is instruction 3 and the program state tells that the current PC matches the instruction's address. As a result, the condition is evaluated to be `false` and the program state becomes $\{\text{PC} : 4; \text{x} : 10\}$. Now, PVTA looks at the final instruction in the path, which is at address 5. However, PC in the program state has value 4, which does not match the target instruction. Therefore, PVTA stops evaluating the path and reports that the path is safe.

Suspicious Cases. There are 4 situations where PVTA deems a path suspicious. First, a `signal` or `sigaction` library function is invoked in the path, which jumps out of the current path and may potentially lead to other security violations. Second, there is an RTL-unsupported assembly instruction in a path. Third, the register ESP is no longer tracked in the program state due to being assigned with a $\top$ or made to point to a non-stack memory address in the Γ, which indicates a possibility of unexpected stack control convention of the original program. The last case is when a memory access is referring to a constant memory address that is not a code address or data-section address. Paths in this case could still be classified to be potentially unsafe according to the threat model. However, we hope to distinguish paths that stay within the application's scope from those relying on "special" memory addresses, because memory not allocated for the application is less likely to be controlled by the attacker. An example will be provided later.

Optimizations. Moreover, we make sure writing to read-only data sections is not

allowed. If it happens during runtime, the path will be terminated, which may crash a program but not cause the target security violation. Thus, we deem the path safe. After a path is assessed, the security assessment together with the path are stored into another dataset, which we call the assessed path dataset. This is similar to what we do for path discovery. By doing so, one-time assessment can be used for evaluating multiple CFGs, which is critical for our research goal of statistically understanding how CFG precision influences the attack surface.

Examples. We next show three simple example paths that evaluate to safe, potentially unsafe, and suspicious, to demonstrate how PVTA works. All three paths share the same arbitrary write primitive and the same security violation point. The pseudo CFG is presented in Figure 5.3. We assume r1 and r2 are registers that can be set to arbitrary values by the arbitrary write primitive. Thus, all paths start with an empty initial program state Γ (i.e., both registers are mapped to a $\top$). Based on the initial Γ, where `r1` is mapped to $\top$, the path condition at instruction 1, `r1 != 0`, evaluates to $\top$ (i.e., either `True` or `False`), making both branches feasible. Therefore, all the three paths in the CFG are so far valid.

After instruction 1, three paths have different behaviors. Path 1 does an assignment to r2 from r1 at instruction 2. Since r1 is mapped to $\top$ in Γ before instruction 2, r2 is still mapped to $\top$ after evaluating instruction 2, which makes the security violation state `r2 == 2` satisfiable. Thus, path 1 is potentially unsafe. Path 2 and 3 both assign r2 with value 0 at instruction 3, resulting in Γ having a record, "r2: 0". Then, they follow different branches of the conditional branch at instruction 4 to reach the security violation point. Before the conditional branch, Γ tells "r2: 0", so that the path condition `r2 == 0` evaluates to `true`. Therefore, path 3 is invalid and safe. In contrast, path 2 follows the valid branch to reach instruction 5, where r2 is overwritten by a value loaded from a memory address stored in r2. According to Γ, r2 holds value 0 before instruction 5. Thus, the memory address for loading is 0, which may not be valid for a memory read in some cases; and we treat path 2 as suspicious.

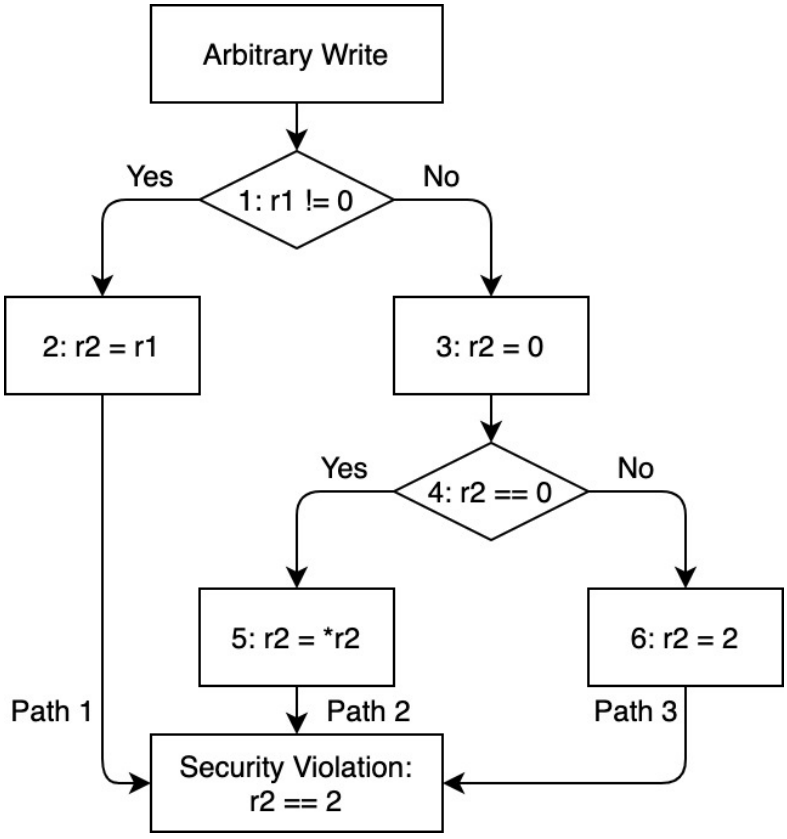


Figure 5.3. An example CFG to show how PVTA evaluates paths.

5.4 Attack Surface Evaluation

The attack-surface evaluation is performed on the assessed path dataset and an input target CFG. The goal of this component is to measure the size of attack surface of the target CFG and to compute how much attack surface is reduced by the target CFG compared to the base CFG. Note that the edges of the target CFG should always be a subset of the base CFG's edge set.

To measure the size of the attack surface, this component determines for every assessed path if it exists in the target CFG; if a path does not belong to the target CFG, the path is illegitimate with respect to the target CFG. Thus, the legitimate risky paths of a target CFG represent the attack surface of the target CFG; and the number of legitimate risky paths measures the attack-surface size. Recall that risky paths are potentially unsafe paths and suspicious paths determined by PVTA. To distinguish from the MazeRunner's attack-surface metric, we define

what SpaceExplorer evaluates as the *attack space* of a program's CFG and the number of legitimate risky paths as the *attack-space size*. A formal definition is given in Definition 2.

Definition 2 (attack space) *A connecting path in a CFG is defined to be a path that connects an arbitrary write primitive to a security-violation point. We write* $ConnPath(V, E, A, S)$ *to be the set of connecting paths in the CFG* $(G = (V, E))$ *given a set of arbitrary write primitives* A *and a set of security-violation points* S*. We represent the process of our per-path security assessment as a Boolean function IsRisky that maps a path to true if and only if the path is determined to be risky (i.e., potentially unsafe or suspicious).*

We define the attack space of a CFG given a set of arbitrary write primitives A *against a set of security-violation points* S*, written as* $ASpace(V, E, A, S)$*, as the set of risky connecting paths. It can be formalized as* $ASpace(V, E, A, S) = \{p \in ConnPath(V, E, A, S) \mid IsRisky(p)\ \}$*. Finally, our attack-space metric is the cardinality of* $ASpace(V, E, A, C)$*.*

However, in practice, it is almost impossible to generate a complete set of connecting paths due to the path explosion problem. We perform a large number of random-walk sampling attempts as discussed in Section 5.2 to acquire an incomplete but representative set. With the representative connecting path set available, the attack-space size can be measured for every target CFG. One straightforward method is to compute the attack-space reduction rate compared to the base CFG as a metric for measuring the security enhancement of CFI. However, the attack-space reduction rate might not be sensitive for comparing different CFI policies, since the number of paths that do not rely on indirect jumps and calls may take the majority of the attack space, while CFI can prevent only attack paths that rely on indirect branches. Therefore, a more sensitive metric for comparing CFI polices should consider only paths involving indirect branches.

Among the three kinds of indirect branches, we further argue that the paths to consider should involve at least one indirect call. First, in our random-walk sampling, we have already assumed a context-sensitive CFI is in place. Thus, all the sampled paths with return instructions but without any indirect calls or jumps are legitimate no matter what CFG is enforced. Moreover, since indirect jumps' targets are resolved in a same way in all CFG construction approaches we consider,

different CFI policies share the same set of legitimate paths that involve indirect jumps but not indirect calls. In all, it is necessary to have at least one indirect call in a path to be possible for CFI to protect.

Therefore, we propose another metric for comparing the security of different CFI policies, we call it the *CFI Security Score*. To compute the metric, we further determine if a path involves an indirect call and compute a reduction rate of indirect-call involved risky paths to score the security improvement of a CFI policy. The formal definition is given in Definition 3.

Definition 3 (CFI Security Score) *We write $ICallConnPath(V, E, A, S)$ to be the set of connecting paths (defined in Definition 2) that contain at least one indirect call instruction, in a CFG ($G = (V, E)$), given a set of arbitrary write primitives A and a set of security-violation points S.*

We define the CFI Security Score for a CFG ($G = (V, E)$) given a set of arbitrary write primitives A against a set of security-violation points S, written as $CFIScore(V, E, A, S)$, as the reduction rate of indirect-call involved risky connecting paths compared to a base CFG ($G_{base} = (V, E_{base})$). It can be formalized as

$$CFIScore(V, E, A, S) = 1 - \frac{|\{p \in ICallConnPath(V, E, A, S) \mid IsRisky(p)\}|}{|\{p \in ICallConnPath(V, E_{base}, A, S) \mid IsRisky(p)\}|}$$

.

5.5 Summary

This chapter presents an unpublished work. The idea is to define attack space by risky paths that are discovered by a per-path value tracking analysis. With all the discovered risky paths, we propose a metric to represent the size of the attack space. We leave the evaluation of this work to Chapter 6, where this metric will be compared with previous metrics.

Chapter 6
Comprehensive Metric Comparison

In this dissertation, we propose two new metrics for measuring the attack surface of a CFI-protected program. One uses risky program points as the attack-surface entities (introduced in Chapter 1), while the other one uses attack paths. Compared to the traditional metrics, our metrics are designed to be more fine-grained. However, the two metrics have their advantages and disadvantages. Therefore, in this chapter, we perform a comprehensive experimental comparison between the AICT metric and our two metrics.

6.1 Comparison Methodology

We use 9 benchmarks from SPEC2006 and 5 security-critical applications to facilitate this comparison. SPEC2006 has a total of 12 benchmarks written in C. However, 3 of them do not contain any indirect calls or any target functions included in our concrete policies, at any CFG precision level; we therefore remove them from our experiments. These 9 benchmarks are mostly for the purpose of demonstrating the functionality and scalability of MazeRunner and SpaceExplorer. In addition, we selected 5 security-critical programs for security evaluation: thttpd-2.29, memcached-1.5.4, lighttpd-1.4.48, exim-4.89, and nginx-1.4.0. `thttpd` is a lightweight ftp/http server; `memcached` is a distributed memory object caching system; `lighttpd` is a web-server program optimized for high-performance; `exim` is a message transfer agent; and `nginx` is a widely used web server. We compiled all programs to x86-32 binaries for evaluation. Our implementations do not sup-

port x86-64 binaries because there is currently no support of translating x86-64 instructions into RTL instructions.

For each benchmark, we use our flexible CFG construction to generate a set of CFGs at different precision levels: address-taken, arity-based, and type-based. Each CFG precision level has its own way of matching indirect calls to target functions. The address-taken CFG matches every indirect call to all address-taken functions, which represents CFI designs such as the original CFI [32] and CCFIR [25]. The arity-based one allows an indirect call to target a function if the number of parameters the function needs matches the number of arguments provided by the call; this is used by TypeArmor [27] and Forward-Edge-CFI [29]. The type-based one corresponds to CFGs used in MCFI [13], PICFI [48], and Newton [8]. Its matching is based on the types of function pointers used in indirect calls and the types of functions. Note that in our experiment, all the CFGs generated for the benchmarks were validated with runtime indirect branch targets.

The three types of CFGs are to acquire ground truth for demonstrating the advantages of our metric compared with traditional graph-based metrics. The ideal ground truth would be a set of all possible real attack paths for each combination of the benchmark program, the CFI defense, the attack model, and the security-violation policy, based on which we could compute the precision and recall. However, it is impractical to acquire such ideal ground truth. Instead, we use accepted knowledge in the literature as indirect ground truth for showing that our metric is more precise than traditional choices. Such knowledge includes:

- **K1**: a higher precision CFG is likely to reduce more attack surface as demonstrated by all CFI systems.
- **K2**: context-sensitivity of CFI protection reduces the number of legitimate paths so that the attack surface should be narrowed down.
- **K3**: the attack surface should be influenced by the capability of attackers and the predefined scope of attacks.

To compare with traditional metrics, we use AICT (average indirect call targets) as a representative. We believe AICT (which considers only indirect calls) is sufficient for measuring CFG precision. Traditionally, all indirect branches are considered for measuring CFG precision. However, all types of CFGs we consider

use the same approach for resolving indirect jumps' targets and return targets are resolved by call graph construction. We note that AICT can reflect only **K1**, because graph-based metrics stay unchanged no matter what capabilities the attacker possesses or what kinds of attacks are considered. Instead, we will show that our metrics can reflect **K1** to **K3**. In all, for each metric, we aim to answer three major questions:

- Is the metric precise enough to reflect the common knowledge (**K1** to **K3**) about CFI and attacks?
- How is the comprehensiveness of the metric?
- What implications does our evaluation have for applying control-flow defenses to real-world applications?

The first question is to understand the precision of our metrics. The second one is similar to computing a recall rate for understanding the chance of generating false negatives. The last one is to show that the new metrics and their underlying analyses are useful for security applications.

6.2 AICT vs MazeRunner

To compare MazeRunner with AICT, we performed evaluation for all combinations of CFG types, attack models, and policies and for all 14 benchmarks. The AICT information and the full set of MazeRunner's 1-step attack-surface measurements are presented in Table 6.1, which are used to compose charts in the following discussions.

6.2.1 Understanding the metric precision

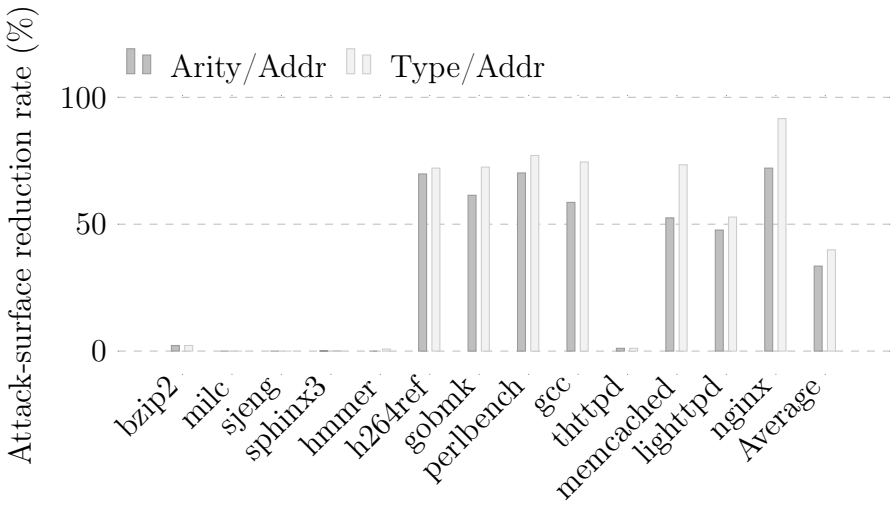
Policy G1 is designed for understanding the attack surface of control-flow hijacking attacks. However, in a data-oriented attack, manipulating control flow is not enough. Passing attacker-desired parameters is also necessary; and G2 is designed for understanding if data flow can be hijacked or not. Thus, G1 and G2 together can measure the attack surface of general data-oriented attacks against CFI defenses. In this section, we use MazeRunner's measurements to confirm the aforementioned

Prog	CFG	AICT	AM0		AM1		AM2	
			G1	G2	G1	G2	G1	G2
bzip2	Addr	2.0	740.0	740.0	721.1	721.1	721.1	721.1
	Arity	1.0	724.0	724.0	705.1	705.1	705.1	705.1
	Type	1.0	724.0	724.0	705.1	705.1	705.1	705.1
milc	Addr	2.0	0.0	806.0	0.0	36.0	0.0	27.0
	Arity	2.0	0.0	806.0	0.0	36.0	0.0	27.0
	Type	2.0	0.0	806.0	0.0	36.0	0.0	27.0
sjeng	Addr	7.0	1069.0	1069.0	1020.0	1020.0	1020.0	1020.0
	Arity	7.0	1069.0	1069.0	1020.0	1020.0	1020.0	1020.0
	Type	7.0	1069.0	1069.0	1020.0	1020.0	1020.0	1020.0
sphinx3	Addr	6.0	898.1	2095.6	432.6	436.1	432.6	427.8
	Arity	5.5	897.0	2093.0	432.0	435.5	432.0	427.3
	Type	5.5	897.0	2093.0	432.0	435.5	432.0	427.3
hmmer	Addr	22.0	329.1	2962.0	161.0	1682.3	161.0	1521.3
	Arity	22.0	329.1	2962.0	161.0	1682.3	161.0	1521.3
	Type	22.0	327.3	2946.0	160.1	1667.2	160.1	1507.1
h264ref	Addr	39.0	2462.2	14894.0	2290.2	14191.5	1319.9	14109.4
	Arity	4.5	803.4	4860.0	691.0	4286.9	398.3	4261.7
	Type	2.7	748.7	4529.0	638.0	3958.6	367.8	3935.2
gobmk	Addr	1786.0	71791.8	58209.5	69253.4	57708.8	48090.7	48091.2
	Arity	866.0	38874.4	31519.8	37208.5	31000.3	23779.2	13427.8
	Type	564.5	27871.9	22598.9	26503.5	22079.4	16939.8	9562.0
perlbench	Addr	721.0	60959.3	78468.9	60892.3	78416.9	57650.1	75847.4
	Arity	122.6	18154.8	23369.5	18123.0	23343.6	17157.6	22582.2
	Type	64.0	13972.9	17986.3	13954.0	17969.0	13211.1	17379.7
gcc	Addr	1216.0	290298.4	439685.5	288169.6	435155.0	280755.4	427562.4
	Arity	405.3	120109.7	181918.0	119347.1	180215.6	116280.8	177137.2
	Type	193.2	74776.6	113256.5	73506.3	110798.0	71598.1	108925.8
thttpd	Addr	17.0	825.0	824.0	749.0	749.0	749.0	749.0
	Arity	8.0	816.0	816.0	741.0	741.0	741.0	741.0
	Type	8.0	816.0	816.0	741.0	741.0	741.0	741.0
memcached	Addr	24.0	824.0	831.9	539.2	764.0	539.2	719.1
	Arity	1.7	356.2	504.6	281.8	409.9	281.8	307.4
	Type	1.5	324.2	500.9	133.0	251.0	133.0	205.6
lighttpd	Addr	54.0	6855.5	6550.8	5728.4	5587.7	5728.4	5362.5
	Arity	12.5	3581.6	3422.4	3008.9	2960.6	3008.9	2794.6
	Type	8.6	3369.3	3184.1	2727.5	2673.0	2727.5	2507.1
nginx	Addr	755.0	182819.7	248112.5	182167.6	247447.9	182167.6	243316.1
	Arity	187.8	50969.8	69173.4	50765.0	68962.9	50765.0	67811.7
	Type	42.0	15829.6	21483.0	15389.8	21020.8	15389.8	20609.7

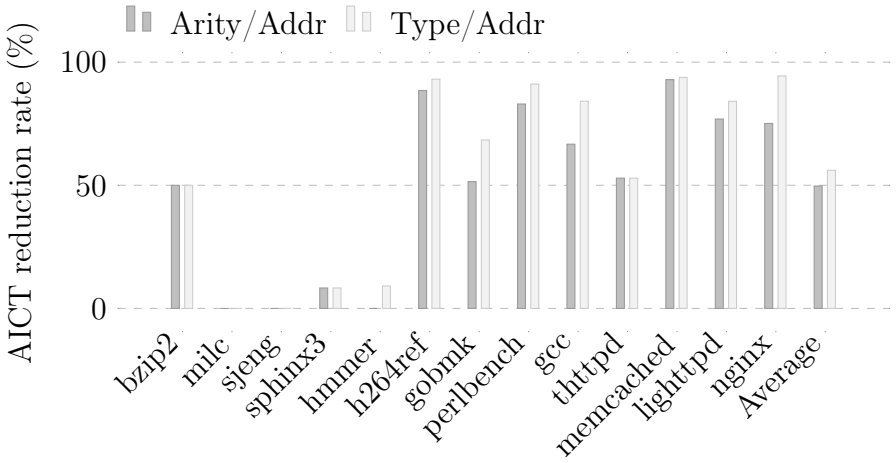
Table 6.1. AICT and 1-step attack-surface measurements for general policies.

accepted knowledge about CFI and attacks, which cannot be reflected by traditional graph-based metrics.

K1: CFG precision improvement may strengthen security. We use the 1-step attack-surface measurements for address-taken CFGs as the bases to compute attack-surface reduction rates for the other two types of CFGs; the larger the



(a) 1-step attack-surface reduction rates for AM2.



(b) AICT reduction rates.

Figure 6.1. Comparison of 1-step attack-surface metric with AICT metric

rate is, the better the CFI's security strength is. We select attack model AM2 as a representative for brevity; and data used for creating Figure 6.1(a) for each benchmark is the average rates yielded from G1 and G2's evaluation. We also do the same reduction rate calculation with the AICT metric and create Figure 6.1(b). Our metric shows that fine-grained CFI reduces a significant attack surface for large programs, including `h264ref`, `gobmk`, `perlbench`, `gcc`, `memcached`, `lighttpd`, and `nginx`, where the type-based approach has the smallest attack surface. This result is consistent with what the AICT metric reflects and conforms with the

accepted knowledge **K1** that fine-grained CFI is likely to be more secure than coarse-grained CFI. However, the difference is that the AICT reduction rates are in general higher than our attack-surface reduction rates, which can be seen by comparing Figure 6.1(a) and Figure 6.1(b).

K2: context-sensitivity improves security. By design, context sensitivity prevents ROP attacks by strictly matching function calls and returns, which reduces a large number of illegitimate paths; in practice, it is implemented through a shadow stack to prevent the manipulation of return addresses. With MazeRunner, we can compare the attack-surface measurements yielded from two attack models, AM0 and AM1, to see how context sensitivity contributes to CFI's security strength, because the difference between AM0 and AM1 is whether the return addresses can be modified by the attacker. To achieve this goal, we compute the reduction rates of the 1-step attack-surface metric from AM0 to AM1 for all benchmarks and present them in Figure 6.2. From the figure, we can observe the attack-surface measurement reduces for most benchmarks for both general policies, with an average of 24.5%; this confirms common knowledge **K2**. However, major reductions are seen in relatively smaller benchmarks. The reason is that large programs can still have a large number of feasible paths even under context-sensitive CFI, making the likelihood of a program point being risky still large. In other words, context-sensitive CFI is less effective in protecting large programs, because the attacker needs only one path to succeed from a large space of paths. In all, our 1-step attack-surface metric, which focuses more on synthesizing the number of risky program points into the attack-surface measurement, still shows the security improvement of context-sensitive CFI, which in contrast cannot be revealed by graph-based metrics.

K3: attackers and attacks should make difference. Another ground truth the traditional metrics cannot reveal is that the attack surface can be different with respect to different attackers and different kinds of attacks. Based on Figure 6.2, we can see that the decrease of attacker's memory capability reduces the attack-surface measurement. In summary, the 1-step attack-surface measurement reduction rate is 24.5% from AM0 to AM1 and 30.8% from AM0 to AM2. At the same time, our attack-surface measurements vary between different security-violation policies. For example, our metric (concrete data can be found in Table 6.1) shows that

`h264ref` has a smaller attack surface for G1 compared with G2 (e.g., 367.8 VS 3935.2 for AM2 in its type-based CFG); by manual inspection, we find that `h264ref` seldom uses the stack or heap for transferring function pointers. In conclusion, our evaluations for different attack models and different policies are consistent with the knowledge about the influences of attackers and attacks (**K3**).

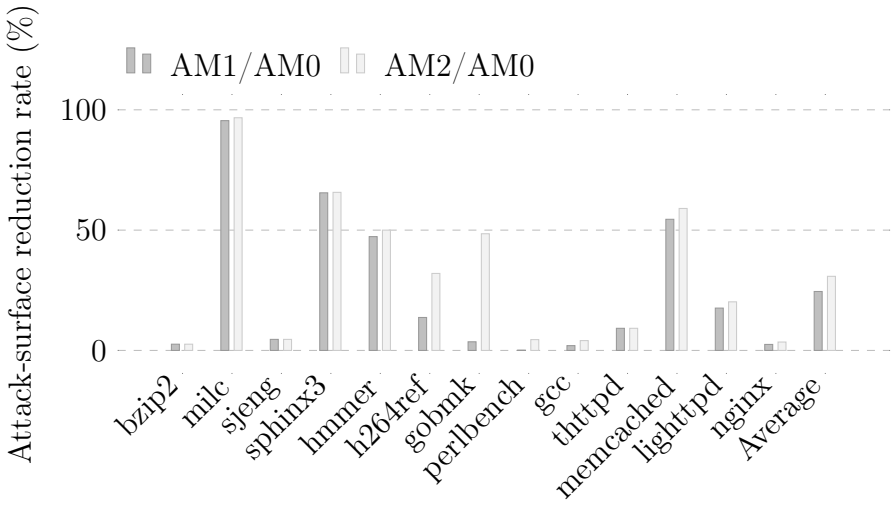


Figure 6.2. Comparison of different attack models for type-based CFGs.

6.2.2 The comprehensiveness of the metric

Without understanding possible false negatives in our static analysis, the precision discussion is not convincing. However, due to the lack of ideal ground truth as we mentioned in our evaluation methodology, it is impossible to evaluate the recall rate. In fact, the same problem exists for traditional metrics; that is, it is challenging to estimate the recall rate for their static analyses. For example, the soundness of a CFG is not easy to demonstrate; still, graph-based metrics are accepted. Nevertheless, we aim to use MazeRunner's evaluations for concrete policies to address the soundness of the attack-aware dependency tracking (ADT) analysis. Target functions in our concrete policies are commonly used in attacks against CFI defenses. We aim to use such concrete policies to show that our attack-surface metric is comprehensive, i.e., known attacks and possible bypasses should be covered by our attack-surface measurement. Among all the attack papers we are aware of [2–10], `nginx` is the most commonly used benchmark. Therefore, we focus on

`nginx` for evaluation with concrete policies. We stress that such evaluations do not prove the soundness of ADT. However, we believe the conservativeness of the analysis together with this evaluation gives evidence to the soundness.

Case-study: arbitrary binary execution. We found 4 attack papers [5, 7, 9, 10] that use `execve` to construct 4 different proof-of-concept (PoC) attacks against `nginx`. In terms of the target CFG precision, [5] assumes a DSA-generated CFG; [7] assumes a fully-precise static CFG; [9] uses an Angr-generated CFG [75–77]; [8, 10] assumes a type-based CFG. The DSA-generated and Angr-generated CFGs are based on static analysis. To be conservative, we assume the CFGs used in the 4 papers are more precise than our type-based CFG. Therefore, one way of validating our metric design is to check that the set of risky program points discovered in the type-based CFG covers all the critical points of the attacks discussed in the 4 attack papers.

Attacks	Critical Funcs	Covered		
		AM0	AM1	AM2
CFB [7]	`ngx_sprintf`	✓	✓	N.A.
	`ngx_exec_new_binary`	✓	✓	N.A.
Control Jujutsu [5]	`ngx_output_chain`	✓	✓	✓
	`ngx_execute_proc`	✓	✓	✓
BOPC [9]	`ngx_execute_proc`	✓	✓	✓
TROP [10]	`ngx_worker_process_exit`	✓	✓	✓
	`ngx_master_process_cycle`	✓	✓	✓
	`ngx_reap_children`	✓	✓	✓
	`ngx_spawn_process`	✓	✓	✓
	`ngx_execute_proc`	✓	✓	✓

Table 6.2. MazeRunner's coverage of critical application functions used in PoC attacks against `nginx`. ✓ means the function is covered by an attack model; N.A. means the related attack is not feasible for the attack model.

The 4 PoC attacks all target at the only direct call site of `execve` in the `ngx_execute_proc` function. For example, BOPC directly uses this function to form a PoC attack, assuming that the whole memory is corrupted right before the basic block that contains the call site. In other words, starting from any predecessor

basic block can result in triggering the security violation. Our ADT gives the same conclusion that all predecessor basic blocks of the `execve` call site are risky. The coverage results are listed in Table 6.2. For each PoC attack, we list the functions used in the attack and show if our risky program points can cover all the functions. In total, there are 8 functions used to construct the 4 PoC attacks. MazeRunner classifies all the basic blocks in the 8 functions as risky under all supported attack models. The only exception is that the attacks in Control-Flow Bending (CFB) [7] are not applicable for AM2, because their attacks cannot defeat a CFI with a shadow stack protecting the return values and AM2 represents an attacker who cannot modify return values.

In conclusion, the risky program points discovered in `nginx` on policy C2 cover all known PoC attacks. The difference is that MazeRunner classifies more risky basic blocks than necessary for the 4 attacks. However, together with the metric precision we have demonstrated, we argue that our overapproximation maintains sufficient precision for making the new metric meaningful while enabling MazeRunner to scale to large applications.

Case study: writable-executable memory region. In `nginx`, there are two direct call sites of `mmap` but none for `mprotect`. Both of the two call sites directly assign a constant value to the third argument, meaning that the attackers following AM0/3/4 have no chance to manipulate the parameter to allocate a writable and executable memory region. Correspondingly, our metric gives 0s as the attack-surface measurements. In this case, CFG precision does not influence the attack surface. The same situation applies to the two call sites of `mmap` in `thttpd`. In all, our evaluation demonstrates that the security violation in policy C1 cannot happen for `nginx`. Our conclusion is different from [8], which constructs a PoC attack by controlling one indirect call twice to target the `malloc` and `mprotect` functions in libc. So, we further check our evaluation of G1 and confirm that the indirect call site is determined by MazeRunner to be controllable.

6.2.3 Implications for applying CFI

Our evaluation has demonstrated the precision and the comprehensiveness of our metric design. Given that our metric is more precise, we conclude that the graph-based metric, using AICT as a representative, can roughly measure the control-flow

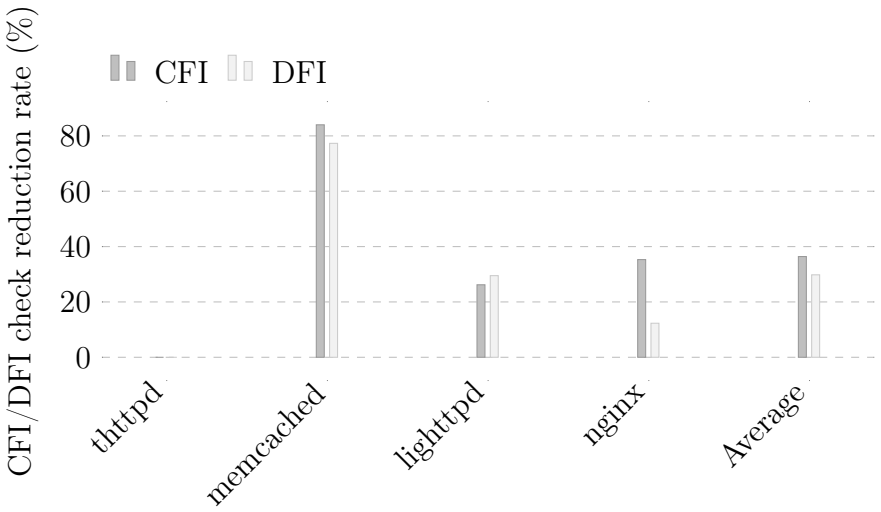


Figure 6.3. CFI and DFI check reduction rates for type-based CFGs.

manipulation space. However, it is not precise enough. A more precise evaluation for the attack surface reduction (i.e., the security benefit of CFI) should consider different attack models and relate to a specific scope of attacks, which is not supported by traditional metrics. Based on our evaluation, we conclude that for large programs and web server applications, where control flow is complex and the demand for security is high, the most fine-grained CFI is recommended. But for small programs, picking a coarse-grained CFI does not lose security compared with fine-grained ones, which can be seen by comparing the attack-surface reduction rates with the AICT reduction rates for `bzip2`, `sphinx3`, `hmmer` and `thttpd` in Figure 6.1.

Furthermore, since an overapproximated metric can be used to prove a security violation is impossible, we use the evaluation result of policy G1 to identify safe indirect calls for which there is no need to insert dynamic checks before them, resulting in better CFI performance. Similarly, the evaluation of policy G2 can be used to discharge unnecessary Data-flow Integrity (DFI) checks for all indirect call arguments. Considering that SPEC2006 benchmarks are not ideal for security evaluations, we present the CFI-check and DFI-check reduction rates only for security-critical benchmarks, w.r.t. the attack model AM0 and type-based CFGs, in Figure 6.3. On average, we can save 36.4% CFI checks and 29.8% DFI checks (the two numbers are 43.8% and 14.5% if all benchmarks are considered). From the results, we can see that there is no reduction of CFI checks for `thttpd`, while there

are significant reductions of CFI checks for `memcached`. After some investigations, we conclude that the reduction rate is highly correlated with the extent of using stack and heap for transferring function pointers. Also, we note that such reductions are safe only if the assumed attack model is more powerful than a real attacker in practice.

6.3 MazeRunner vs SpaceExplorer

In this section, we first demonstrate that SpaceExplorer is able to reflect **K1** to **K3** as what MazeRunner can do. Then, we compare the differences between SpaceExplorer and MazeRunner through experiments. We added `exim` for SpaceExplorer's experiment. For understanding SpaceExplorer's precision and code coverage, we use both SPEC2006 benchmarks and the 5 security-critical programs. However, for demonstrating security applications of SpaceExplorer, we use the security-critical programs only.

6.3.1 SpaceExplorer's precision

K2 is obvious for SpaceExplorer. Though the path finding algorithm of SpaceExplorer directly assumes context-sensitive CFI, if the assumed CFI was not context-sensitive, it is straightforward that the number of paths would increase significantly. Thus, risky paths would increase accordingly. Next, we apply SpaceExplorer on benchmarks and different policies to show **K1** and **K3**.

SpaceExplorer assumes an attack model that requires one arbitrary write primitive to launch attacks. We design SpaceExplorer to take as input a set of arbitrary write primitives. Ideally, the arbitrary write primitives should come from known CVEs that may lead to arbitrary write primitives. However, for some benchmarks, such CVEs are not available. Even if there exist such CVEs for some programs, the number of the CVEs is limited to only a few, which might not be sufficient for evaluating the overall attack surface. Thus, in addition to user-specified arbitrary write primitives, we provide another choice to randomly sample a percentage of basic blocks as the arbitrary write primitives for an overall evaluation of the attack surface. We call the percentage, **AW-ratio**.

In addition to the general policies defined in Section 4.2.3, we further consider

a new general policy, where the security violations are memory operations that could lead to arbitrary memory write/read vulnerability. For example, a memory write through a register and a memory read from an indexed memory address are candidates. However, we do not want to make all candidates to be security violations; otherwise, the number of connecting paths could easily become too large. Thus, we also sample a user-specified percentage of candidate memory operations as security violation points. We call this policy **G3: AWrite** and the percentage **SV-ratio**.

In experiment, we set a 0.5% SV-ratio to construct the G3 policy for each benchmark. We further set a 0.5% AW-ratio for every benchmark to create pseudo arbitrary write primitives. For sampling paths, we gave each security violation point 50000 random-walk attempts with 500 basic blocks as the depth for each attempt. G1's security violation points are all the indirect calls, while G3's security violation points are the randomly selected 0.5% of memory operations that may lead to arbitrary memory write/read vulnerabilities. Note that the random-walk sampling is performed backwardly starting from security violation points. The number of sampled paths for each benchmark is listed in the second columns of Table 6.3 and Table 6.4, for G1 and G3 policies respectively. Such paths constitute the data set for yielding the following experimental conclusions.

SpaceExplorer reflects K1 and K3. In Table 6.3 and Table 6.4, we also present reduction rates of 5 different candidate metrics for each benchmark to support the proposal of CFI security score. The first grouped columns show the C-Type CFG's reduction rates of different metrics compared to the base CFG, i.e., the Address-Taken CFG. The "AICT" column is for the average indirect call target reduction; the "Total" column shows the reduction rates of the total connecting paths (i.e., paths connect an arbitrary write primitive and a security violation); the "TotalRisky" column lists the reduction rates of the connecting paths deemed risky by PVTA (i.e., the reduction of attack space as defined in 2); the "ICall" column shows the reduction rates of indirect-call involved connecting paths; and the "ICallRisky" column is for the reduction rates of indirect-call involved risky connecting paths (i.e., the CFI security score as defined in 3). The second grouped columns show the same categories of reduction rates of the Arity CFG.

For `milc`, neither the C-Type CFG nor the Arity CFG reduce the AICT. Thus,

the two CFGs are identical to the base CFG and have 0.0 reduction rates of all the four kinds of metrics. For `sphinx3`, all the sampled paths do not involve indirect calls, making the "ICall" and "ICallRisky" metric not calculable. Both situations apply to `sjeng`: C-Type and Arity CFGs do not reduce AICT and the only sampled path does not involve indirect call. For other benchmarks, when C-Type's AICT reduction is higher than Arity's AICT reduction, all the four other reductions are also higher, which corresponds to the **K1** knowledge (i.e., smaller AICT leads to smaller attack surface). The CFI security scores (in the "ICallRisky" column) are different for G1 and G3 policies, which shows the **K3** knowledge (i.e., the attack goal influences the attack surface).

Another observation from the table is that the reductions of risky connecting paths ("TotalRisky" column) are consistently smaller than the CFI security scores ("ICallRisky" column), which supports our motivation of proposing the CFI security score. Moreover, one may consider using the paths that involve indirect calls to approximate the attack surface, so to avoid the cost of evaluating paths with PVTA. However, as we can see from the table, "ICall" and "ICallRisky" metrics may vary a lot for some benchmarks. Thus, we believe PVTA based security assessment is necessary.

Prog	Paths	Ctype Reduction %					Arity Reduction %				
		AICT	Total	TotalRisky	ICall	ICallRisky	AICT	Total	TotalRisky	ICall	ICallRisky
bzip2	24729	46.5	35.9	25.1	67.2	56.0	46.5	35.9	25.1	67.2	56.0
milc	15399	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
hmmer	27040	9.0	1.4	2.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
h264ref	3965172	93.0	95.9	97.6	99.3	99.6	88.4	95.9	97.6	99.3	99.6
sphinx3	443	64.7	1.1	3.9	nan	nan	64.7	1.1	3.9	nan	nan
sjeng	1	0.0	0.0	0.0	nan	nan	0.0	0.0	0.0	nan	nan
gobmk	207397	68.4	75.7	77.9	90.8	89.0	51.5	69.4	71.6	86.4	84.9
perlbench	741476	95.4	72.1	75.1	98.2	98.0	83.7	67.6	70.6	93.1	93.3
gcc	1601217	86.8	61.0	69.5	94.5	95.8	66.9	44.1	50.7	72.8	76.3
nginx	2045689	76.1	70.6	78.0	80.6	82.6	62.3	62.5	69.6	74.2	75.8
exim	255102	61.9	34.8	34.4	61.7	62.0	56.6	33.0	33.0	59.9	60.4
memcached	911582	94.0	80.9	93.6	99.1	99.6	92.7	80.6	93.4	98.8	99.5
lighttpd	296867	84.0	76.6	53.8	98.2	95.9	76.7	74.8	48.4	96.0	86.1
thttpd	4638	45.0	0.0	0.0	-nan	-nan	45.0	0.0	0.0	-nan	-nan

Table 6.3. ICall Policy with Ctype and Arity CFGs

SpaceExplorer discovers real attack paths. We construct a demo program that is vulnerable to control-flow hijacking attacks, which is shown in Listing 6.1. In the program, the `main` function parses a command-line input and a file input. The command-line input determines the sorting method to be used and the file input contains the byte sequence to be sorted, which simulates a chunk of data

Prog	Paths	Ctype Reduction %					Arity Reduction %				
		AICT	Total	TotalRisky	ICall	ICallRisky	AICT	Total	TotalRisky	ICall	ICallRisky
bzip2	50289	46.5	1.2	1.8	54.6	74.0	46.5	1.2	1.8	54.6	74.0
milc	30927	0.0	0.0	0.0	nan	nan	0.0	0.0	0.0	nan	nan
hmmer	143696	9.0	2.8	1.4	22.4	16.0	0.0	0.0	0.0	0.0	0.0
h264ref	968865	93.0	25.4	24.4	95.2	98.1	88.4	25.3	24.2	94.5	97.0
sphinx3	42261	64.7	3.3	3.5	25.6	24.5	64.7	3.3	3.5	25.6	24.5
sjeng	350811	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
gobmk	4902919	68.4	57.5	63.7	87.2	89.9	51.5	46.4	52.0	70.3	73.5
perlbench	5687694	95.4	53.4	57.0	96.9	96.5	83.7	49.0	52.5	90.2	90.2
gcc	6702694	86.8	25.0	36.2	93.6	95.6	66.9	20.2	29.7	78.0	81.0
nginx	3807504	76.1	62.1	67.1	83.0	81.0	62.3	54.4	59.9	75.2	74.8
exim	482264	61.9	18.4	20.2	70.9	63.9	56.6	17.3	19.3	66.8	59.8
memcached	469240	94.0	56.3	52.6	95.5	96.7	92.7	53.5	51.9	93.0	95.5
lighttpd	673365	84.0	43.5	58.3	88.7	92.5	76.7	40.2	54.3	82.1	86.2
thttpd	130969	45.0	2.1	4.5	68.0	89.8	45.0	2.1	4.5	68.0	89.8

Table 6.4. AWrite Policy with Ctype and Arity CFGs

received over the network. However, in the program, there is a buffer-overflow vulnerability at line 40 that may overwrite the local function pointer `sort` to point to a `mprotect` wrapper function. When the execution reaches line 43, the condition can never be satisfied, because `strlen` can only return positive integers. So, the function pointed to by `mp2` can never be executed, even though the function call seems dangerous. Then, the `sort` function pointer is invoked. However, due to the buffer-overflow vulnerability, this pointer may point to a `mprotect` wrapper function. According to the C1: wx-mem policy introduced in (Section 4.2.3), to make `mprotect` sensitive, the attacker needs to pass an integer that has "110" or "111" as the last three digits as the third argument. By carefully crafting the input byte sequence to be of a proper length, the security violation condition can be satisfied.

```
1  #include ...
2
3  typedef void (*sortmethod)(int, char*);
4  typedef void (*mprot1)(int);
5  typedef void (*mprot2)(int, void*);
6  typedef void (*mprot3)(void*, int);
7
8  void up(int size, char* arr) {
9    for(int i = 0; i < size - 1; i++) {
10     for(...) { if(...) {...} } // iteratively swapping
11   }
12 }
13 void down(int size, char* arr) {...} {
```

```
14   for(int i = 0; i < size - 1; i++) {
15     for(...) { if(...) {...} } // iteratively swapping
16   }
17 }
18
19 void mprotect_wrapper1(int prot)
20   { mprotect(up, 4096, prot); }
21 void mprotect_wrapper2(int prot, void* start)
22   { mprotect(start, 4096, prot); }
23 void mprotect_wrapper3(void* start, int prot)
24   { mprotect(start, 4096, prot); }
25
26 mprot1 mp1 = mprotect_wrapper1;
27 mprot2 mp2 = mprotect_wrapper2;
28 mprot3 mp3 = mprotect_wrapper3;
29
30 int main (int argc, char** argv) {
31   sortmethod sort = NULL;
32   char method[10], input[20];
33   ... // some code for initialization
34   strncpy(method, argv[1], 10);
35   if (!strncmp(method, "up", 10)) sort = up;
36   else if (!strncmp(method, "down", 10)) sort = down;
37   else exit(0);
38
39   FILE* fp = fopen("input.bin", "rb");
40   fread(input, sizeof(int), 20, fp);
41   fclose(fp);
42
43   if(strlen(input) < 0) mp2(6, input);
44   sort(strlen(input), input);
45
46   return 0;
47 }
```

Listing 6.1. Buffer-Overflow Vulnerable Program

After enforcing CFI, the number of attack paths is reduced. In the base CFG, there is an attack path connecting the memory corruption site (line 40) to every security violation point (line 20, 22, or 24), because the base CFG allows `sort` to point to all three `mprotect` wrappers ((`mprotect_wrapper1`, `mprotect_wrapper2`,

and `mprotect_wrapper3`). However, `mprotect_wrapper3` is not guaranteed to cause security violation, because it uses the second parameter, which can only be a memory address passed in by the caller `sort` whose concrete value is not controlled by the attacker, to determine the third argument of `mprotect`. In the Arity CFG, `mprotect_wrapper1` is no longer a legitimate target, so only `mprotect_wrapper2` can cause security violation. In the C-Type CFG, neither `mprotect_wrapper1` nor `mprotect_wrapper3` is reachable, but `mprotect_wrapper2` is kept and is still causing security violation. In total, there are 2 attack paths for the base CFG and 1 attack path for both Arity CFG and C-Type CFG.

Other than the attack paths, there are safe paths. For example, without considering path conditions, `mp2` is reachable in a CFG. Thus, depending on what targets are allowed by the CFG, `mp2` may target different functions, resulting in more paths to `mprotect` callsites. In all, there are 39 paths that can connect the memory corruption site at line 40 to a `mprotect` callsite in the Base CFG. In detail, 3 paths skip line 43 and reach line 44 to invoke a `mprotect` wrapper through function pointer `sort`. 3 paths reach line 43 to invoke the 3 `mprotect` wrappers through function pointer `mp2`. $(4 \times 3 = 12)$ paths first go to function `up` at line 43, then return to line 44 to reach to the 3 `mprotect` wrappers. Similarly, there are 12 paths that pass function `down` and reach all the `mprotect` wrappers. Note that such 24 paths are the result of avoiding multiple iterations of a loop; otherwise, there would be an infinite number of paths. At last, function pointer `mp2` can also target at all the `mprotect` wrappers. After returning from those wrappers, at line 44, the paths go to such wrappers again, thus creating another $(3 \times 3 = 9)$ paths.

Security Assessment	Base	Arity	Ctype
Groud-Truth-Total	39	24	11
SpaceExplorer-Total	39	24	11
Groud-Truth-Safe	36	22	10
SpaceExplorer-Safe	29	20	8
Groud-Truth-Unsafe	2	1	1
SpaceExplorer-Unsafe	10	4	3

Table 6.5. Ground Truth Security Assessment and SpaceExplorer's Evaluation

So far, we have the ground-truth security assessment of the paths in the base CFG of the program, where there are 39 paths with 2 of them are attack paths. We do the same analysis and we can know the total paths for different CFGs: 24 for

Arity and 11 for Ctype. The ground-truth security assessment and SpaceExplorer's evaluation output on this program are presented in Table 6.5. SpaceExplorer took 1.658s to finish the evaluation. Comparing the first two rows, we can see that SpaceExplorer covers all the paths in the program. Comparing the second two rows, among all the safe paths, SpaceExplorer can identify most safe paths. The last two rows demonstrate that SpaceExplorer is conservative in determining the security of a path; thus, it can be used to reduce the number of candidates for symbolic execution and runtime verification in automatic exploit generation. In detail, among the 10 unsafe paths of the base CFG, there are 2 invoking `mprotect` through `sort` and another 2 invoking `mprotect` by mp2. The rest 6 (3×2) unsafe paths pass the 3 wrappers via `mp2` and reach the 2 `mprotect` callsites through `sort`.

One may question that how about paths that pass function `up` and `down` via `mp2` and reach the 2 `mprotect` callsites through `sort`. There should exist risky paths among them but not discovered by SpaceExplorer. The reason is that the maximum allowed loop iterations was set to be 1 during path sampling. The sampled paths that pass function `up` and `down` can at most incur one iteration of the loops in these two functions. However, the first argument of `mp2` is a constant 6, which must incur 5 iterations of the top-level loops in function `up` and `down`. Thus, all sampled paths that pass function `up` and `down` are safe, because the path conditions are not feasible. However, if we allowed more than 5 iterations of loops, with sufficient attempts of random-walk sampling, SpaceExplorer would find risky paths that pass through function `up` and `down`, even though the paths are not real attack paths due to the unsatisfiable path condition at line 43.

Based on SpaceExplorer's evaluation of the demo program, we summarize that SpaceExplorer can discover real attack paths with false positives and false negatives. The major source of false positives is the overapproximation of the validity of path conditions. Overapproximation results in infeasible risky paths, as demonstrated by the 2 risky paths invoking `mprotect` wrappers through `mp2` and the 6 paths passing the callees of `mp2`. As a future direction, we can use value sets [38] to replace the $\top$ value to reduce overapproximation. As for false negatives, it is obvious that the incompleteness of path sampling is the major source.

6.3.2 SpaceExplorer's coverage

Due to the design of MazeRunner, it achieves a complete coverage of the whole program. That is, every program point is given a security assessment despite significant overapproximation. However, SpaceExplorer randomly samples paths in the program, which may suffer from limited code coverage. Therefore, we demonstrate SpaceExplorer's code coverage of path sampling for all the programs and both G1 and G3 policies in Table 6.6. We measure the code coverage by checking if a basic block is involved in at least one sampled path. Since SpaceExplorer aims at finding connecting paths, the number of arbitrary write points (AW) and the number of security violation sites (SV) influence the number of paths that can be discovered. We also present the two numbers in the table for reference. Note that the AW number is given as 0.5% of all basic blocks in the program, while SV for G1 is the number of indirect calls and for G3 is 0.5% of the program's non-constant memory writes. The median code coverage of G1/G3 is 41.1%/71.2%. G3 has a much higher code coverage because of the larger amount of security violation points: the more security violations there are, the easier to find a connecting path.

Prog	AW	G1: Target Control		G3: AWrite	
		SV	Coverage %	SV	Coverage %
bzip2	15	20	34.6	80	71.2
milc	24	4	2.9	48	55.5
hmmer	69	9	12.5	238	32.9
h264ref	99	369	42.3	538	71.9
sphinx3	37	8	6.6	94	34.6
sjeng	31	1	0.1	88	84.1
gobmk	185	44	43.5	454	81.2
perlbench	319	139	66.7	735	87.9
gcc	1032	474	43.2	2106	65.8
nginx	211	414	78.2	622	83.1
exim	198	89	39.8	250	67.2
memcached	37	75	66.8	100	71.1
lighttpd	64	122	54.9	139	72.2
thttpd	19	1	18.5	33	46.0

Table 6.6. SpaceExplorer's Code Coverage

6.3.3 Security application of SpaceExplorer

In general, automatic exploit generation (AEG) requires symbolic execution to compute a malicious input to make sure an attack path is feasible. The malicious input is by tradition called payload. Then, the payload is fed to the program during runtime to verify the attack path. To discover an attack path, AEG needs to statically traverse the program to discover candidate paths for the symbolic execution to run with. Therefore, the key is to quickly locate an attack path to feed to the symbolic execution. However, the number of paths is in most cases exponential to the number of basic blocks, which poses a huge challenge for AEG work to "intelligently" reduce the candidates for trying symbolic execution. In our case, though SpaceExplorer is not designed to automatically generate or verify a payload, it can efficiently determine safe paths to reduce the burden of symbolic execution, which eventually would increase the efficiency of AEG. In Table 6.7, we present for each security-critical benchmark's different CFGs, how many safe paths can be discharged. The medians are presented in the last two rows, based on which we summarize that SpaceExplorer can reduce on average 70.6% (the average of all medians) of the symbolic execution tasks for automatic exploit generation.

Prog	Policy	Base		Arity		Ctype	
		Total	Safe %	Total	Safe %	Total	Safe %
nginx	G1	2045689	58.4	766942	66.3	600914	68.9
	G3	3807504	67.9	1734845	71.7	1441247	72.1
exim	G1	255102	62.6	171027	62.6	166437	62.3
	G3	482264	67.1	398862	67.9	393753	67.8
memcached	G1	971582	53.8	188036	84.3	185345	84.5
	G3	469240	61.6	218172	60.3	205167	58.3
lighttpd	G1	296867	89.4	74919	78.4	69440	79.1
	G3	673365	78.9	402608	83.8	380639	84.4
thttpd	G1	4638	81.6	4638	81.6	4638	81.6
	G3	130969	72.7	128223	73.4	128223	73.4
Median	G1	-	62.6	-	78.4	-	79.1
	G3	-	67.9	-	67.9	-	67.8

Table 6.7. The percentage of safe paths determined by SpaceExplorer

6.4 Statistical Relation Between AICT and Attack Surface

Since AICT is simple to compute, we hope to understand at what situation AICT is still meaningful for measuring the security improvement of enforcing a CFI policy. So, we do a statistical analysis of the relation between attack surface and AICT for the 5 security-critical benchmarks. To achieve this goal, for each benchmark, we compute 50 uniformly distributed intermediate AICT values between the base CFG's AICT and C-Type CFG's AICT. For each intermediate AICT value, we randomly generate 10 CFG variations; they vary in the details of indirect-call target set but share the same AICT measurement. In other words, the CFGs are different but are deemed equivalent by the AICT metric. Since the base CFG and C-Type CFG of `thttpd` are the same, we cannot generate CFGs of different precision levels. Thus, `thttpd` is excluded for this experiment. We apply SpaceExplorer and MazeRunner to the rest of benchmarks with G1 as the policy, because G1 is the policy implicitly assumed by AICT. In Figure 6.4, we plot the relation between the metric reduction and AICT reduction. The blue dots are for SpaceExplorer's metric; red dots are for MazeRunner's metric (assuming attack model AM2); and the gray dots are for AICT itself.

First, the reduction rate of MazeRunner's metric is always smaller than SpaceExplorer's metric. The reason is that CFI has less influence under MazeRunner's attack model than under SpaceExplorer's attack model. MazeRunner's attack model assumes multiple-time memory corruptions in each path. The attacker can almost corrupt memory every time before an indirect call to manipulate its target. In this case, CFI cannot eliminate any attack paths that reach the call, making a large portion of program points to be risky for the indirect call. In contrast, SpaceExplorer assumes a one-time memory corruption. To manipulate an indirect call far from the memory corruption site, the attacker needs to shape the initial memory state to accommodate the original data flows and path constraints. Thus, SpaceExplorer's metric relying on risky paths is more sensitive to the validity of a path, on which CFI has a huge impact. Therefore, when AICT improves (i.e., the CFG precision improves), SpaceExplorer's metric reduces more than MazeRunner's metric.

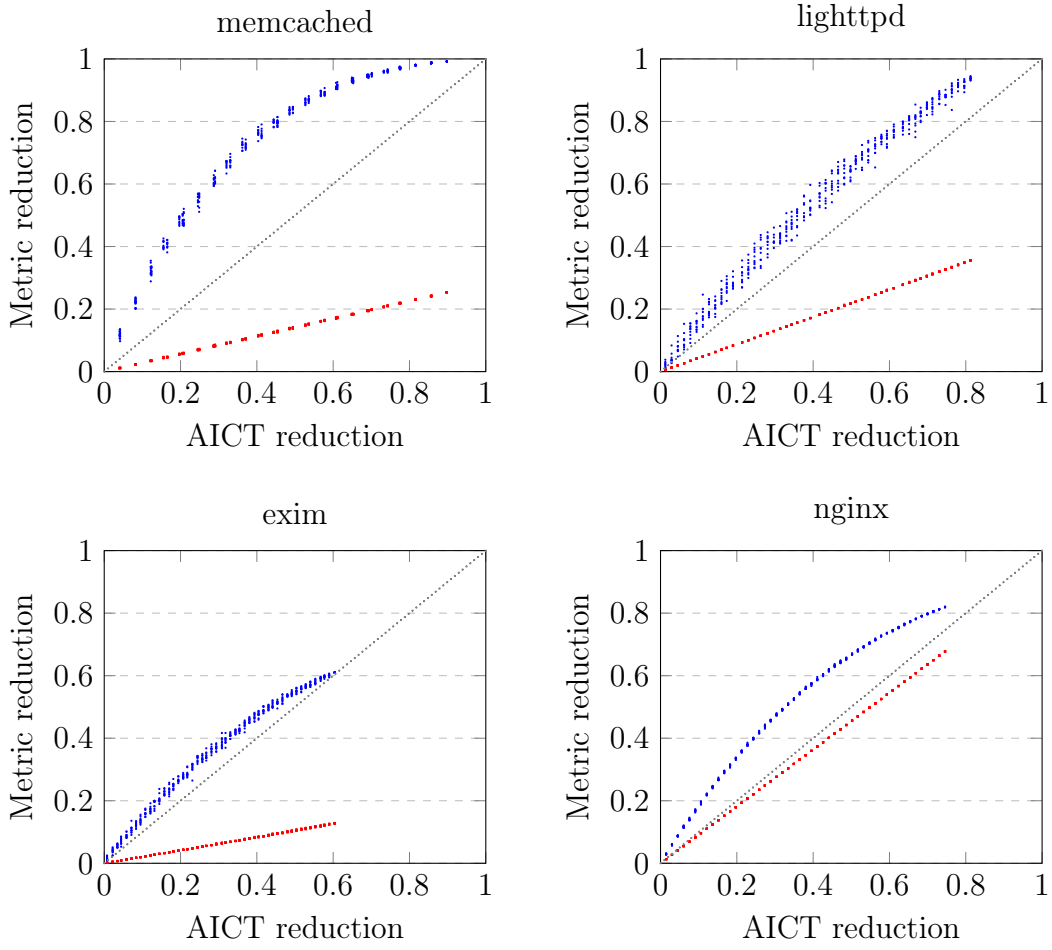


Figure 6.4. Statistical relations between metrics and AICT with G1; blue dots are for SpaceExplorer's metric; red dots are for MazeRunner's metric; gray dots are for AICT.

Second, we observe that MazeRunner's metric reduction is almost linear to AICT reduction, while SpaceExplorer's metric reduction presents a trend to slow down when AICT reduction is reaching the limit. We argue that SpaceExplorer gives a more reasonable curve than MazeRunner. We give an informal explanation. When AICT is large, removing one target from an indirect call's target set is likely to result in more paths reduced than when AICT is small. Further, a larger amount of path reduction indicates a larger probability of eliminating attack paths. Therefore, as the AICT reduction approaches the limit, the number of attack paths that can be incrementally prevented should be fewer. Therefore, the attack surface reduction is likely to be slower, which is shown by SpaceExplorer's metric.

Third, AICT is insensitive to the details of CFGs. On the contrary, both

SpaceExplorer and MazeRunner are sensitive. However, we find that SpaceExplorer is more sensitive than MazeRunner. For each AICT reduction rate, we have 10 CFG variations; thus, SpaceExplorer and MazeRunner can produce 10 different measurements at each AICT reduction rate. According to the Figure 6.4, we can see that SpaceExplorer's metric is more sensitive to the details of CFG than MazeRunner's metric. On average, SpaceExplorer has a 5.0% fluctuation (i.e., the largest score minuses the smallest score) at an AICT level, while MazeRunner has only a 0.1% fluctuation. The reason is mostly due to the difference in the granularity of the attack-surface entity. When the CFG details change, the valid paths change accordingly, while reachable program points might not change significantly, because there would often be alternative paths that connect two points in a program.

Prog	Instrs	ICall	BaseAICT	MinAICTRed
memcached	35552	79	22.8	4.5%
lighttpd	61504	126	52.3	9.7%
exim	168602	92	83.2	5.0%
nginx	240652	417	749.6	0.0%

Table 6.8. Minimum Expected AICT Reductions for G1

Due to the observation of metric variations given the same AICT, we claim that improving AICT does not necessarily lead to security improvement, unless the AICT improvement is significant enough. Thus, an interesting question would be how much is enough. To answer this question, for each of the 8 selected benchmarks, we compute an AICT reduction rate that guarantees an attack-surface reduction (in terms of SpaceExplorer's metric) for policy G1. We call such an AICT reduction the minimum expected AICT reduction and list the related data in Table 6.8, where the minimum expected AICT reduction is abbreviated as MinAICTRed in the table. We observe that as the program size and complexity increase, the minimum expected AICT reduction becomes smaller. Based on our experimental data, we conclude that AICT improvement should be around 4.8% (the median of the data points we have) to be trustworthy to support a new CFI policy. However, due to the lack of a large set of benchmarks, we cannot conclude a statistical relation between the program size and the minimum expected AICT reduction, which we leave as a possible future direction.

6.5 Summary

This chapter presents experiments that demonstrate the improvement of the proposed metrics over the classic graph-based metric, the average indirect call target (AICT). We first demonstrate the improved precision and comprehensiveness of MazeRunner's metric compared to AICT. Moreover, due to MazeRunner's overapproximation of attack surface, it can be used to discharge CFI/DFI runtime checks to reduce performance overhead. In the second step, we show that SpaceExplorer not only can achieve what MazeRunner can do in terms of quantitatively evaluating the attack surface, but also can help discover real attacks. Since SpaceExplorer may suffer from incompleteness of path sampling, we compute the code coverage statistics to evaluate the completeness. We also perform a statistical analysis to understand the relation between AICT and attack surface. We find that SpaceExplorer's metric is more informative than MazeRunner's metric and AICT reduction should be around 3.2% to be trustworthy for claiming a better CFI policy.

Chapter 7
Future Work

7.1 Framework Extension

The current framework does not support binaries compiled from C++ programs nor x64 binaries. The bottleneck is the limitation of our CFG construction tool [15]. Extending the CFG construction tool is non-trivial; C++ code has its own patterns of indirect branches, such as virtual calls, and x64 has a different instruction set.

7.1.1 Extension to x64

Our current infrastructure cannot be simply migrated to support x64. Our disassembly module is built in Coq with proofs for x86 only. Adding support for x64 would also require rewriting the proofs, which itself is an interesting topic for formal method research. To avoid such complexity, we propose to use existing binary reverse engineering tools, such as BAP and Angr, for disassembly. Then, we translate their results into our format to fit in our type inference and meta-information based CFG construction. In particular, we model the assembly instructions with our abstract model of x64; we parse their CFGs into our format as base CFGs for type inference and flexible CFG construction; and we translate their modeling of instruction semantics into RTL instructions for our attack surface evaluation.

7.1.2 Extension to C++

To deal with C++ programs, the CFG construction requires new methodologies to deal with C++'s type system. First, function overloading allows one function

name to have different implementations. During our type analysis, we need to use debugging information to distinguish different implementations for the same function name. Second, we need to consider the "class" type in our type inference module. Classes introduce another layer of complexity of type inference. Three access specifiers make the field accesses in classes different from field accesses of C struct types; some are through direct variable access, while some are through member functions. Inheritance and virtual functions make type-signature matching more complicated, because the inheritance determines the actual type signature of a callsite. To solve this problem, we want to propose a binary-level class hierarchy analysis relying on compiler-generated meta-information.

7.2 Automatic Exploit Generation

To utilize SpaceExplorer's efficient determination of safe paths, we aim to improve the efficiency of the state-of-the-art automatic exploit generation tool based on SpaceExplorer. First, we plan to employ reinforcement learning in path discovery. Second, we need to involve lightweight symbolic execution to generate seed input. Then, we feed the seed input to a runtime verifier to check if the input triggers security violation. At last, the attack verification is used as feedback to the reinforcement learning. There are some challenges. We need to carefully balance the efficiency and accuracy of symbolic execution. The attack verification must be fast and produce informative feedback. And the last challenge is designing the reward function for the Reinforcement-Learning agent.

7.3 Typed Binary-level Alias Analysis

To construct binary-level CFGs, there are two major threads of binary-level approaches: signature matching and alias analysis. The signature matching approach relies on type information to infer signatures; alias analysis performs expensive points-to analysis to infer targets of indirect branches. People may have thought about merging the two approaches' final results to a CFG and enforce it with CFI. However, we propose to use type information during alias analysis to improve the synthesis of the two approaches. Type information helps improve the granularity of memory modeling and can also compensate for the precision loss caused by flow

insensitivity of alias analysis. It also has its own challenges, such as how to assign type information to binary-level, how to design the memory modeling with type information and how to enforce the type system in a static analysis. Moreover, with the capability of discharging CFI checks from MazeRunner, the typed binary-level alias analysis can lead to a better CFI.

7.4 Automatic Program Generation

In program analysis related research, one common difficulty is to acquire a high-quality benchmark set with ground truth. Manually selected or created benchmarks often suffer from limitations such as incompleteness, impracticality, and even bias. Borrowing the idea of Generative Adversarial Network, we propose to research on the topic of automatic test program generation. Program analysis is trying to discover ground truths among a huge amount of interference introduced by the programming language. With ground truth, can we automatically generate a program that contains the ground truth but introduce interference that fail the program analysis tool? In fact, program obfuscation belongs to this direction. We could consider to use dynamic analysis on some initial benchmarks to collect ground truth and employ machine/deep learning to generate variations of the original benchmarks.

Chapter 8
Conclusion

To evaluate the security strength of CFI policies, our first step is to propose an approach for high-precision CFG construction, without compiler modification, to generate the policies of CFI. The approach uses compiler-generated meta-information to retrieve source-level information for CFG construction. It relies on a type-inference engine that deduces types of indirect-branch operands from source-level types in debugging information. Our system is compatible with multiple compilers and multiple compiler versions, thanks to its compiler-independent design. Also, it is customized to produce CFGs of different precision levels for the security evaluation.

We propose MazeRunner, a framework for quantitatively evaluating the attack surface of a CFI-protected program. In contrast to traditional metrics, our metric provides an overapproximated estimation and considers how gadgets can be chained to form an attack path. We propose a novel attack-aware dependency tracking for a fine-grained attack-surface evaluation, in which attacker's influences are considered. Moreover, a point-to-stack analysis is employed for simplifying memory modeling and enabling the partitioning of memory to support different attack models. Since our attack models are relatively stronger than real-world attackers and our system is designed to overapproximate, our metric is conservative but meaningful for measuring the insecurity of a program. Our experiments demonstrated precision improvement of the new metric than the traditional graph-based metrics. Based on our metric, we confirm the desire for fine-grained CFI and context-sensitivity in large and critical applications. Also, our evaluation can discharge unnecessary 43.8% CFI checks and 14.5% DFI checks for parameters on average.

While MazeRunner is able to give an overapproximated estimation of the attack surface, it cannot construct any attack paths. Thus, we are motivated to improve

the attack-surface granularity to paths and to assume a less powerful attack model for a more precise evaluation of the attack surface, resulting in a new evaluation framework, SpaceExplorer. We employ a random-walk based path sampling method to discover candidate paths to address the problem of path explosion. To check if a path is risky, we propose a novel per-path value tracking analysis to conservatively determine the insecurity of the path. Given a CFG, we can measure how many risky paths are reduced compared to a baseline CFG to measure the security improvement. We design the whole process into a pipeline structure to optimize the performance. In experiment, we demonstrated that SpaceExplorer achieves good code coverage and better accuracy in terms of measuring the attack surface. Moreover, SpaceExplorer can efficiently determine safe paths so that it can help improve the performance of automatic exploit generation.

In all, by constructing CFGs and demonstrating the feasibility of two quantitative metrics for measuring the attack surface, we conclude that *the attack surface of a CFI-protected program can be measured by fine-grained and scalable quantitative metrics.*

Bibliography

[1] Morrisett, G., G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan (2012) "RockSalt: Better, Faster, Stronger SFI for the x86," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 395–404.

[2] Göktas, E., E. Athanasopoulos, H. Bos, and G. Portokalidis (2014) "Out of Control: Overcoming Control-Flow Integrity," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 575–589.

[3] Davi, L., A.-R. Sadeghi, D. Lehmann, and F. Monrose (2014) "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *23rd Usenix Security Symposium*, pp. 401–416.

[4] Carlini, N. and D. Wagner (2014) "ROP is Still Dangerous: Breaking Modern Defenses," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, USENIX Association, USA, p. 385–399.

[5] Evans, I., F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos (2015) "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity," in *22nd ACM Conference on Computer and Communications Security (CCS)*, pp. 901–913.

[6] Conti, M., S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi (2015) "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *22nd ACM Conference on Computer and Communications Security (CCS)*, ACM, pp. 952–963.

[7] Carlini, N., A. Barresi, M. Payer, D. Wagner, and T. R. Gross (2015) "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *24th Usenix Security Symposium*, pp. 161–176.

[8] van der Veen, V., D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia (2017) "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *24th ACM Conference on Computer and Communications Security (CCS)*, ACM, pp. 1675–1689.

[9] Ispoglou, K. K., B. AlBassam, T. Jaeger, and M. Payer (2018) "Block oriented programming: Automating data-only attacks," in *25th ACM Conference on Computer and Communications Security (CCS)*, ACM, pp. 1868–1882.

[10] Farkhani, R. M., S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi (2018) "On the Effectiveness of Type-based Control Flow Integrity," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ACM, pp. 28–39.

[11] van der Veen, V., D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida (2015) "Practical Context-Sensitive CFI," in *CCS15*, CCS '15, Association for Computing Machinery, New York, NY, USA, p. 927–940.
URL `https://doi.org/10.1145/2810103.2813673`

[12] Zhang, M. and R. Sekar (2013) "Control Flow Integrity for COTS Binaries," in *22nd Usenix Security Symposium*, pp. 337–352.

[13] Niu, B. and G. Tan (2014) "Modular Control-Flow Integrity," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 577–587.

[14] Burow, N., S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer (2017) "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys*, **50**(1), pp. 16:1–16:33.

[15] Zeng, D. and G. Tan (2018) "From Debugging-Information Based Binary-Level Type Inference to CFG Generation," in *8th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 366–376.

[16] Muntean, P., M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert (2019) "LLVM-CFI: Analyzing Static Control Flow Integrity Protections," in *Annual Computer Security Applications Conference (ACSAC)*, pp. 584–597.

[17] Lu, K. and H. Hu (2019) "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in *26th ACM Conference on Computer and Communications Security (CCS)*, pp. 1867–1881.

[18] Kim, S. H., C. Sun, D. Zeng, and G. Tan (2021) "Refining Indirect Call Targets at the Binary Level," in *Network and Distributed System Security Symposium (NDSS)*, The Internet Society.

[19] Li, Y., M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu (2020) "Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms,"

in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, Association for Computing Machinery, New York, NY, USA, p. 1821–1835.
URL https://doi.org/10.1145/3372297.3417867

[20] Avgerinos, T., S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley (2014) "Automatic Exploit Generation," *Commun. ACM*, **57**(2), p. 74–84.
URL https://doi.org/10.1145/2560217.2560219

[21] Cha, S. K., T. Avgerinos, A. Rebert, and D. Brumley (2012) "Unleashing Mayhem on Binary Code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, IEEE Computer Society, USA, p. 380–394.
URL https://doi.org/10.1109/SP.2012.31

[22] Hu, H., Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang (2015) "Automatic Generation of Data-Oriented Exploits," in *24th Usenix Security Symposium*, pp. 177–192.

[23] Bao, T., R. Wang, Y. Shoshitaishvili, and D. Brumley (2017) "Your exploit is mine: Automatic shellcode transplant for remote exploits," in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 824–839.

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

[25] Zhang, C., T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou (2013) "Practical Control Flow Integrity and Randomization for Binary Executables," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 559–573.

[26] Niu, B. and G. Tan (2013) "Monitor Integrity Protection with Space Efficiency and Separate Compilation," in *20th ACM Conference on Computer and Communications Security (CCS)*, CCS '13, p. 199–210.

[27] van der Veen, V., E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida (2016) "A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 934–953.

[28] Andriesse, D., X. Chen, V. van der Veen, A. Slowinska, and H. Bos (2016) "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries," in *25th Usenix Security Symposium*, pp. 583–600.

[29] Tice, C., T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike (2014) "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, USENIX Association, USA, p. 941–955.

[30] Pewny, J. and T. Holz (2013) "Control-Flow Restrictor: Compiler-based CFI for iOS," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, Association for Computing Machinery, New York, NY, USA, p. 309–318.
URL `https://doi.org/10.1145/2523649.2523674`

[31] Wang, Z. and X. Jiang (2010) "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 380–395.

[32] Abadi, M., M. Budiu, Ú. Erlingsson, and J. Ligatti (2005) "Control-flow integrity," in *12th ACM Conference on Computer and Communications Security (CCS)*, pp. 340–353.

[33] Ge, X., N. Talele, M. Payer, and T. Jaeger (2016) "Fine-Grained Control-Flow Integrity for Kernel Software," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 179–194.

[34] Zeng, D., B. Niu, and G. Tan (2021) "MazeRunner: Evaluating the Attack Surface of Control-Flow Integrity Policies," in *The 20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2021)*, IEEE.

[35] Caballero, J. and Z. Lin (2016) "Type Inference on Executables," *ACM Computing Surveys*, **48**(4), pp. 65:1–65:35.

[36] Lin, Y. and D. Gao (2021) "When Function Signature Recovery Meets Compiler Optimization," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 36–52.

[37] Kruegel, C., W. Robertson, F. Valeur, and G. Vigna (2004) "Static Disassembly of Obfuscated Binaries," in *13th Usenix Security Symposium*, pp. 255–270.

[38] Balakrishnan, G. and T. Reps (2004) "Analyzing Memory Accesses in x86 Executables," in *International Conference on Compiler Construction (CC)*, pp. 5–23.

[39] Wartell, R., Y. Zhou, K. W. Hamlen, and M. Kantarcioglu (2014) "Shingled Graph Disassembly: Finding the Undecidable Path," in *Proceedings*

of the 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), Tainan, Taiwan, pp. 273–285.

[40] Wartell, R., Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham (2011) "Differentiating Code from Data in x86 Binaries," in *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, vol. 3, pp. 522–536.

[41] McCamant, S. and G. Morrisett (2006) "Evaluating SFI for a CISC Architecture," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USENIX Association, USA.

[42] Yee, B., D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar (2009) "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *2009 30th IEEE Symposium on Security and Privacy*, pp. 79–93.

[43] Sehr, D., R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen (2010) "Adapting Software Fault Isolation to Contemporary CPU Architectures," in *19th Usenix Security Symposium*, pp. 1–12.

[44] Erlingsson, Ú., M. Abadi, M. Vrable, M. Budiu, and G. Necula (2006) "XFI: Software Guards for System Address Spaces," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 75–88.

[45] Akritidis, P., C. Cadar, C. Raiciu, M. Costa, and M. Castro (2008) "Preventing Memory Error Exploits with WIT," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 263–277.

[46] Davi, L., R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A. reza Sadeghi (2012) "MoCFI: A framework to mitigate control-flow attacks on smartphones," in *Network and Distributed System Security Symposium (NDSS)*, The Internet Society.

[47] Niu, B. and G. Tan (2014) "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity," in *21st ACM Conference on Computer and Communications Security (CCS)*, pp. 1317–1328.

[48] ——— (2015) "Per-Input Control-Flow Integrity," in *22nd ACM Conference on Computer and Communications Security (CCS)*, pp. 914–926.

[49] Payer, M., A. Barresi, and T. R. Gross (2015) "Fine-Grained Control-Flow Integrity Through Binary Hardening," in *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, DIMVA 2015, Springer-Verlag, Berlin, Heidelberg, p.

144–164.
URL https://doi.org/10.1007/978-3-319-20550-2_8

[50] Ding, R., C. Qian, C. Song, B. Harris, T. Kim, and W. Lee (2017) "Efficient protection of path-sensitive control security," in *26th Usenix Security Symposium*, pp. 131–148.

[51] Khandaker, M., A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng (2019) "Adaptive Call-Site Sensitive Control Flow Integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 95–110.

[52] Khandaker, M. R., W. Liu, A. Naser, Z. Wang, and J. Yang (2019) "Origin-sensitive Control Flow Integrity," in *28th Usenix Security Symposium*, USENIX Association, Santa Clara, CA, pp. 195–211.

[53] Kuznetsov, V., L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song (2014) "Code-Pointer Integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 147–163.

[54] Abera, T., N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik (2016) "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *CCS'16*, pp. 743–754.

[55] Dessouky, G., S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A. Sadeghi (2017) "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware," in *DAC'17*, pp. 24:1–24:6.

[56] Zeitouni, S., G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A. Sadeghi (2017) "ATRIUM: Runtime attestation resilient under memory attacks," in *ICCAD'17*, pp. 384–391.

[57] Dessouky, G., T. Abera, A. Ibrahim, and A. Sadeghi (2018) "LiteHAX: lightweight hardware-assisted attestation of program execution," in *ICCAD'18*, ACM, p. 106.

[58] Abera, T., R. Bahmani, F. Brasser, A. Ibrahim, A. Sadeghi, and M. Schunter (2019) "DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems," in *NDSS'19*.

[59] Liu, J., Q. Yu, W. Liu, S. Zhao, D. Feng, and W. Luo (2019) "Log-Based Control Flow Attestation for Embedded Devices," in *Cyberspace Safety and Security - 11th International Symposium, CSS'19, Part I*, vol. 11982 of *Lecture Notes in Computer Science*, Springer, pp. 117–132.

[60] Nunes, I. D. O., S. Jakkamsetti, and G. Tsudik (2020) "Tiny-CFA: A Minimalistic Approach for Control-Flow Attestation Using Verified Proofs of Execution," *CoRR*, **abs/2011.07400**.

[61] Toffalini, F., E. Losiouk, A. Biondo, J. Zhou, and M. Conti (2019) "ScaRR: Scalable Runtime Remote Attestation for Complex Systems," in *RAID'19*, pp. 121–134.

[62] Hu, J., D. Huo, M. Wang, Y. Wang, Y. Zhang, and Y. Li (2019) "A Probability Prediction Based Mutable Control-Flow Attestation Scheme on Embedded Platforms," in *TrustCom/BigDataSE'19*, pp. 530–537.

[63] Sun, Z., B. Feng, L. Lu, and S. Jha (2020) "OAT: Attesting Operation Integrity of Embedded Devices," in *SP'20*, IEEE, pp. 1433–1449.

[64] Huo, D., Y. Wang, C. Liu, M. Li, Y. Wang, and Z. Xu (2020) "LAPE: A Lightweight Attestation of Program Execution Scheme for Bare-Metal Systems," in *22nd IEEE International Conference on High Performance Computing and Communications; 18th IEEE International Conference on Smart City; 6th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2020*, IEEE, pp. 78–86.

[65] Zhang, Y., X. Liu, C. Sun, D. Zeng, G. Tan, X. Kan, and S. Ma (2021) "ReCFA: Resilient Control-Flow Attestation," in *Annual Computer Security Applications Conference*.

[66] Xu, X., M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin (2019) "CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software," in *28th USENIX Security Symposium (USENIX Security 19)*, USENIX Association, Santa Clara, CA, pp. 1805–1821. URL `https://www.usenix.org/conference/usenixsecurity19/presentation/xu-xiaoyang`

[67] Zeng, D. and G. Tan (2018) "From Debugging-Information Based Binary-Level Type Inference to CFG Generation," in *8th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 366–376.

[68] Jang, D., Z. Tatlock, and S. Lerner (2014) "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," in *Network and Distributed System Security Symposium (NDSS)*, The Internet Society.

[69] Pawlowski, A., M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida (2017) "MARX: Uncovering class Hierarchies in C++ Programs," in *Network and Distributed System Security Symposium (NDSS)*.

[70] Menapace, J., J. Kingdon, and D. MacKenzie (1999) *The "stabs" debug format*.

[71] DWARF Debugging Information Format Committee (2017) *DWARF Debugging Information Format Version 5*.

[72] Niu, B. (2015) *Practical Control-Flow Integrity*, Ph.D. thesis, Lehigh University, Bethlehem, PA.

[73] Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood (2005) "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 190–200.

[74] Schneider, F. (2000) "Enforceable Security Policies," *ACM Transactions on Information and System Security*, **3**(1).

[75] Shoshitaishvili, Y., R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna (2016) "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 138–157.

[76] Stephens, N., J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna (2016) "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Network and Distributed System Security Symposium (NDSS)*, The Internet Society.

[77] Shoshitaishvili, Y., R. Wang, C. Hauser, C. Krügel, and G. Vigna (2015) "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," in *Network and Distributed System Security Symposium (NDSS)*, The Internet Society.

Vita

Dongrui Zeng

Education

- **PhD Candidate in Computer Science**, advised by Prof. Tan, Gang
  - Pennsylvania State University, University Park, PA; 1/2016-12/2021
  - Lehigh University, Bethlehem, PA; 8/2014-12/2015
- **B.S. in Computational Mathematics**
  - Nanjing University, Nanjing, China; awarded in 7/2014

Professional Experience

- **Security Research Engineer Intern**
  - Palo Alto Networks, Santa Clara, CA; 5/2021-8/2021

Publications During Ph.D. Study

- Zhang, Y.; Liu, X.; Sun, C.; **Zeng, D.**; Tan, G.; Kan X.; and Ma S. (2021). ReCFA: Resilient Control-Flow Attestation. In *The 2021 Annual Computer Security Applications Conference (ACSAC)*.
- **Zeng, D.**; Niu, B.; and Tan, G. (2021). MazeRunner: Evaluating the Attack Surface of Control-Flow Integrity Policies. In *20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*.
- Kim, S. H.; Sun, C.; **Zeng, D.**; and Tan, G. (2021). Refining Indirect Call Targets at the Binary Level. In *The Network and Distributed System Security Symposium (NDSS)*.
- (Liu, S. and **Zeng, D.**); Huang, Y.; Capobianco, F.; McCamant, S.; Jaeger, T.; and Tan, G. (2019). Program-mandering: Quantitative Privilege Separation. In *26th ACM Conference on Computer and Communications Security (CCS)*. **Co-first author**.
- **Zeng, D.** and Tan, G. (2018). From debugging-information based binary-level type inference to CFG generation. In *8th ACM Conference on Data and Application Security and Privacy (CODASPY)*. **Outstanding paper award**.