1-2021

Novel techniques in recovering, embedding, and enforcing policies for control-flow integrity

Yan LIN
Singapore Management University

NOVEL TECHNIQUES IN RECOVERING, EMBEDDING, AND ENFORCING POLICIES FOR CONTROL-FLOW INTEGRITY

YAN LIN

SINGAPORE MANAGEMENT UNIVERSITY

2021

Novel Techniques in Recovering, Embedding, and Enforcing Policies for Control-Flow Integrity

by

Yan Lin

Submitted to School of Computing and Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

Singapore Management University
2021

I hereby declare that this PhD dissertation is my original work
and it has been written by me in its entirety.
I have duly acknowledged all the sources of information
which have been used in this dissertation.

This PhD dissertation has also not been submitted for any degree
in any university previously.

[illegible]

Yan Lin

22nd January, 2021

Novel Techniques in Recovering, Embedding, and Enforcing Policies for Control-Flow Integrity

Yan Lin

Abstract

Control-Flow Integrity (CFI) is an attractive security property with which most injected and code-reuse attacks can be defeated, including advanced attacking techniques like Return-Oriented Programming (ROP). CFI extracts a control-flow graph (CFG) for a given program and instruments the program to respect the CFG. Specifically, checks are inserted before indirect branch instructions. Before these instructions are executed during runtime, the checks consult the CFG to ensure that the indirect branch is allowed to reach the intended target. Hence, any sort of control-flow hijacking would be prevented.

There are three fundamental components in CFI enforcement. The first component is accurately recovering the policy (CFG). Usually, the more precise the policy (CFG) is, the more security CFI improves, but precise CFG generation was considered hard without the support of source code. The second one is embedding the CFI policy securely. Current CFI enforcement usually inserts checks before indirect branches to consult a read-only table which stores the valid CFG information. However, this kind of read-only table can be overwritten by some kinds of attacks (e.g., Rowhammer attack and data-oriented programming). The third component is to efficiently enforce the CFI policy. In current approaches, no matter whether there are attacks, the CFI checks are always executed whenever there is an indirect control-flow transfer. Therefore, it is critical to minimize the performance impact of the CFI checks.

In this dissertation, we propose novel solutions to handle these three fundamental components. We systematically study how compiler optimization would impact CFG recovery by investigating two methods that recover CFI policy based on func-

tion signature matching at the binary level and propose our novel improved mechanism to more accurately recover function signature. We also propose an enhanced deep learning approach to recover function signature by including domain-specific knowledge to the dataset. To embed CFI policy securely, we design a novel platform which encodes the policy into the machine instructions directly without relying on consulting any read-only data structure by making use of the idea of instruction-set randomization. In it, each basic block is encrypted with a key derived from the CFG. To efficiently enforce CFI policy, we make use of a mature dynamic code optimization platform called DynamoRIO to enforce the policy so that it only requires to do the CFI check when needed.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my supervisor Associate Professor Debin GAO for his guidance in my research and help me develop strong research skills. I am also very grateful to the other dissertation committee members Prof Robert DENG, Prof David LO and Prof Zhenkai LIANG for taking time reviewing my thesis and providing valuable suggestions. Their comments help me clarify my thesis, refine my approach and make me become a more rigorous researcher.

Also, I acknowledge the friendship and support from my group members in the security group of SMU. I would like to thank the following university staffs: Pei Huan Seow, Yar Ling Yeo and Chew Hong Ong, for their unfailing support and assistance.

Finally, I would like to thank my parents and sisters, who are always supporting me and encouraging me with all their best wishes.

List of Publications

Conference Papers

Yan Lin and Debin Gao. When Function Signature Recovery Meets Compiler Optimization. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, USA, 2021.

Yan Lin,Xiaoyang Cheng, and Debin Gao. Control-Flow Carrying Code. In *Proceedings of the 14th ACM ASIA Conference on Computer and Communications Security*, New Zealand, 2019.

Xiaoyang Cheng, **Yan Lin**, and Debin Gao. DynOpVm: VM-based Software Obfuscation with Dynamic Opcode Mapping. In *Proceedings of the 17th International Conference on Applied Cryptography and Network Security*, Colombia, 2019.

Xiaoxiao Tang, **Yan Lin**, Daoyuan Wu, and Debin Gao. Towards Dynamically Monitoring Android Applications on Non-rooted Devices in the wild. In *Proceedings of the 11th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, Sweden, 2018.

Yan Lin, Xiaoxiao Tang, and Debin Gao. SafeStack+: Enhanced Dual Stack to Combat Data-Flow Hijacking. In *Proceedings of the 22nd Australasian Conference on Information Security and Privacy*, Australia, 2017.

Jianming Fu, Rui Jian and **Yan Lin**. FRProtector: Defeating Control Flow Hijacking Through Function-level Randomization and Transfer Protection. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks*, Canada, 2017

Xiaoxiao Tang, Yu Liang, Xinjie Ma, **Yan Lin**, and Debin Gao. On the Effectiveness of Code-reuse-based Android Application Obfuscation. In *Proceedings of the 19th Annual International Conference on Information Security and Cryptology*, Korean, 2016.

Yan Lin, Xiaoxiao Tang, Debin Gao, and Jianming Fu. Control Flow Integrity Enforcement with Dynamic Code Optimization. In *Proceedings of the 19th Information Security Conference*, USA, 2016.

Chapter 1

Introduction

In this chapter, we first introduce concepts and implementations of Control-Flow Integrity [3], which is a fundamental approach to mitigating control-flow hijacking attacks, and then present practical issues of previous CFI systems and summarize how we address those problems.

1.1 Overview of Control-Flow Integrity

Application is often written in memory-unsafe languages; this makes it prone to memory errors that are the primary attack vector to subvert systems. Many protection mechanisms including DEP (Data Execution Prevention [4]), ASLR (Address Space Layout Randomization [83]), and GS/SSP (Stack Smashing Protector [25]) have gained wide adoption, and they are making it more difficult for attackers to exploit vulnerabilities. These mechanisms can mitigate various standard attacks, but these reactive defenses can often be bypassed by advanced exploitation techniques [64, 68].

Natural protection against control-flow hijacking attacks is to enforce CFI (Control-Flow Integrity [3]). The goal of CFI is to restrict the set of possible control-flow transfers to those that are strictly required for correct program execution. This prevents control-flow hijacking attacks such as Return-Oriented Pro-

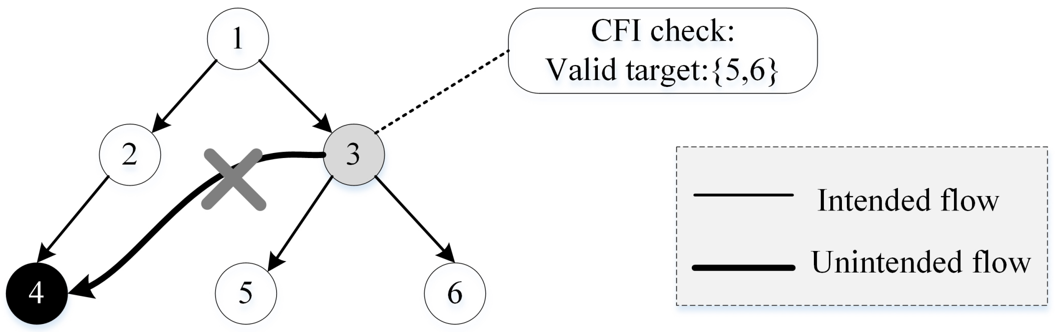


Figure 1.1: Example of Control-Flow Integrity

gramming (ROP) [10, 17, 73] from working because they would cause the program to execute control-flow transfers, which are illegal under CFI. Figure 1.1 shows a high-level representation of CFI: first, a Control-Flow Graph (CFG), which approximates the set of legitimate control-flow transfers is construct prior to program execution [97, 94, 57, 58]. Next, a CFI check is inserted for indirect branches (e.g., indirect calls, indirect jumps, and returns). These checks ensure that all executed branches correspond to edges in the CFG at runtime. For instance, the valid targets of node 3 can only be either 5 or 6. If the adversary aims to redirect execution to node 4, CFI will immediately terminate the program execution.

Despite CFI's efficacy, it has not seen wide adoption. We believe that not well supporting some critical features contributes to CFI's poor deployment.

First of all, having an accurate CFI policy (CFG) is known to be hard as it is generally difficult to identify the target locations for all control-flow transfers. Most binary-level CFI techniques [97, 3, 94] have to conservatively consider all functions as potential targets of an indirect caller, resulting in loosened CFI policies which make them vulnerable to various attacks [27, 36, 71]. Most fine-grained approaches require the availability of the source code [57, 58, 84], so that they can construct a more precise CFG by making use of type information available in the source code. This kind of information is not available in binary since compilers do not preserve much language-level information in the process of compilation. Two fine-grained approaches TypeArmor [85] and τCFI [55] are proposed to recover a fine-grained CFG by matching the function signature (the number of arguments and argument width) at indirect caller and callee sites at the binary level. However, they rely on strictly following the calling convention used by compilers, and various compiler optimizations may violate the calling conventions. For example, modern compilers typically do not (re)set the argument registers explicitly at the caller site if the intended value is already in the corresponding register. It would result in incorrect identification of the number of arguments and/or argument widths.

In addition, existing CFI approaches [3, 97, 85] use memory page protection mechanism, Data Execution Prevention (DEP) as an underlying basis. Therefore, they can use read-only tables to store valid targets of indirect branches [97] and insert read-only tags inside the code segment [3]. At runtime, these tables and tags will be checked to see whether the execution follows the policy. However, there are scenarios in which such page-level protection is unavailable, e.g., bare-metal systems which do not have a Memory Management Unit (MMU) and applications with dynamically generated code. Moreover, data race attacks [96], Rowhammer attacks [11], and Data-oriented programming (DOP) [40] have demonstrated that it is possible to gain arbitrary memory read and write access.

Furthermore, CFI-protected programs require extra execution time and space compared to their native counterparts [97, 3, 94]. For example, whenever there is an indirect control-flow transfer, the CFI checks are executed no matter whether there are attacks, thus the protected programs are in general slower than the native versions. For instance, classic CFI reports about 20% performance overhead which hinders its wide adoption.

1.2 Thesis Statement

Given these challenges, an interesting question is: does a CFI solution that supports fine-grained CFGs, independence of memory page protection mechanism, and efficiency exist?

Thesis Statement: *Control-Flow Integrity can be fine-grained, independent of memory page protection mechanism, and efficient.*

In this dissertation, we propose approaches to well support these three features. We first systematically study the practicality of recovering fine-grained CFI policies for binaries compiled with different optimization levels. Next, we propose C^3, a novel CFI approach which encodes CFI policies into the machine instructions directly without relying on the assumption that read-only data and code cannot be overwritten. Last but not least, we propose *DynCFI* to enforce CFI based on the dynamic code optimization platform DynamoRIO [12] to improve the performance. The details of these works are presented as follows.

1.3 Practicality of Recovering Fine-grained CFI Policies

Since having an accurate CFI policy (CFG) is the prerequisite to CFI enforcement, in this dissertation, we do a systematic study on the practicality of recovering fine-grained CFI policies at the binary level. Specifically, we study how compiler optimization would impact the accuracy of CFG construction (function signature recovery) on x86-64 platform. Specifically, we first theoretically analyze the possible ways in which compiler optimizations could impact the accuracy of two most recent approaches in function signature recovery for CFI, namely TypeArmor [85] and τCFI [55], and then experiment with a large number of testing binaries to evaluate the extent to which such complications arise on real-world applications. All testing binaries are obtained by using two commonly used compilers: gcc-8 and clang-7, with different optimization levels ranging from O0 to O3 for x86-64.

The result shows that compiler optimizations have both positive and negative impacts on function signature recovery. For example, optimizations make the identification of variadic functions more accurate. However, compile optimizations could make identification of the number of arguments and the type inferencing at callees less accurate, because of the elimination of unused arguments and promotion/demotion of argument types. In order to mitigate these inaccuracies, we propose our improved policies to recover the function signatures more accurately. We also propose an enhanced deep learning approach to recover function signature by including domain-specific knowledge to the dataset.

1.4 Control-Flow Carrying Code

A novel CFI approach called C^3 which encodes CFI policies into the machine instructions directly without relying on the assumption that read-only data and code cannot be overwritten is implemented. In it, each basic block in the program is encrypted with a key derived from the CFG. More specifically, the key is derived from the addresses of valid callers of the basic block to ensure correct control-flow transfers. At runtime, only the valid callers (their addresses) could enable the correct reconstruction of the key to decrypt the basic block. The challenge is a basic block may have multiple valid callers, while the successor block has to be encrypted with a single key. In order to enable the reconstruction of the single correct key by all the valid control-flow transfers, secret sharing scheme [75] is used to make the key shared among valid callers.

We apply C^3 to a number of server and non-server applications on the Linux platform. Our experimental results demonstrate that C^3 effectively defends against control-flow hijacking attacks and at the same time, introduces realistic runtime performance overhead for server applications comparable to existing Instruction-Set Randomization (ISR) implementations on the same instrumentation platform.

1.5 Control-Flow Integrity Enforcement Based on Dynamic Code Optimization

A framework that can efficiently enforce CFI based on dynamic code optimization platform is implemented. A lot of well established and mature dynamic code optimizers are proven to introduce minimal overhead, and we believe that they could result in a system that significantly outperforms existing CFI implementation. We enforce a set of security policies on top of DynamoRIO [12] for CFI properties. The results show that it can achieve better performance compared with previous CFI approaches. Moreover, we further investigate the exact contribution to this performance improvement. Specifically, we propose a three-dimensional design space and perform comprehensive experiments to evaluate the contribution of each axis in the design space of performance overhead. The results show that traces in the dynamic optimizer had contributed the most performance improvement. This is because the trace mechanism can avoid some indirect branch lookups by inlining a popular target of an indirect branch into a trace.

1.6 Organization

The reminder of this dissertation is organized as follows: Chapter 2 is a literature review which examines closely related research. Chapter 3 presents details on the systematic evaluation on the extent to which compiler optimization could impact the accuracy of existing approaches in function signature recovery, and then an enhanced deep learning approach to recover function signature is proposed. Chapter 4 describes the system C^3 that embeds the CFI policy into machine instructions. Chapter 5 introduces the framework *DynCFI* that enforce CFI with a dynamic code optimization platform. Chapter 6 summarizes the contribution of this thesis.

Chapter 2

Literature Review

2.1 Control-Flow Hijacking

C and C++ are perhaps the most important programming languages due to its high performance. However, these programs all suffer from memory corruption issues such as out-of-bound accesses or object use-after-free bugs.

Listing 2.1 shows a real memory corruption bug found in `MiniUPnP` 1.0. An attacker has full control over `action` at line 6. The code at line 10 copies `methodlen` size of data from the address pointed by `p` to the address pointed by `method`. Hence, if the attacker passes a long quoted method (more than 2048 bytes), it will cause buffer overflow, which results in code-injection and code-reuse attacks.

```
ExecuteSoapAction(struct upnphttp *h, const char *action,, int n)
{
  char *p;
  char method[2048];
  ...
  p = strchr(action,'#');
  methodlen = strchr(p,'"') - p - 1;
  ...
  memset(method, 0, 2048);
  memcpy(method, p, methodlen);
  ...
}
```

Listing 2.1: A stack buffer overflow bug (CVE-2013-0230) in MiniUPnP 1.0

Code-injection Attack. In code injection attacks, the attacker injects new code into the address space of the victim program and executes her code. In the example of Listing 2.1, if the stack is executable, the attacker could inject the shellcode to the `method` array and the return address of the current function can also be overwritten with a pointer pointing to the entry of the injected code. Then, when the current function returns, instead of returning to the caller, the function returns to the injected code and executes it. This kind of control-flow hijacking attack has been mitigated by some effective mechanisms, such as Data Execution Prevention [4] and Instruction-set randomization [42, 7, 65].

Code-reuse Attack. In code-reuse attacks, rather than directly injecting shellcode to the victim, the attacker chains the existing code bytes (gadgets) together to perform the malicious operation, such as bypassing DEP, so that she can execute the injected shellcode. In the example in Listing 2.1, the attacker could simply overwrite the return address to the address of a libc function `system` and write arguments to the function on the stack. When the function returns, `system` will be executed with attacker-fed arguments, which enable the attacker to execute arbitrary commands.

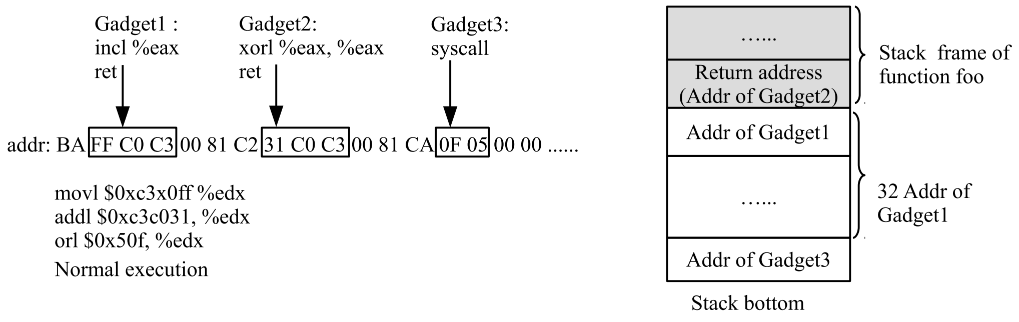


Figure 2.1: A proof of concept example of Return-Oriented Programming attacks

Usually, simply overwriting a return address is not sufficient to mount an attack. Attackers use a more advanced code reuse technique called Return-Oriented Programming (ROP) [73] to perform any malicious operation. To mount such an attack, the attackers first scan the code bytes and find gadgets, which are instruction sequences ending with a return (or an indirect call/jump), and perform basic operations such as an addition or memory load. Then, by carefully overflowing the stack (heap), the attackers can chain these gadgets into an arbitrary program.

We use a simple proof-of-concept example in Figure 2.1 to demonstrate principles of ROP attacks in x86-64 Linux. Suppose function `foo` has a stack overflow vulnerability as shown in Listing 2.1, so the attacker can overwrite the stack buffer. As shown, there is a sequence of instructions which perform some basic arithmetic operations at address `addr`. However, if we decode the instruction from the middle, three ROP gadgets will be found. Gadget1 increments register %eax by one and returns; Gadget2 sets %eax to zero; and Gadget3 performs a system call. If the attacker overwrite the return address of function `foo` to the address of Gadget2, and write 32 copies of Gadget1 addresses, followed by the address of Gadget3, the attackers can essentially set %eax to 0, increment it by 32, and execute a system call after `foo` returns. Since Linux uses %eax to pass the system call number to the kernel, and 32 is the system call number for pause, the current process will sleep.

A lot of mitigation techniques are deployed to defend against this kind of attack, including stack canaries and Address Space Layout Randomization (ASLR).

2.2 Deployed Defenses

Stack Canaries. Stack canaries [25] mitigate control-flow hijacking attacks by monitoring the integrity of return addresses. It inserts a canary before every return address and checks the value of the canary before a function returns. If the canary changes, stack overflow might have happened and the program is terminated. However, stack canary can be bypassed. As shown in BROP [9], since the cookie is a randomly chosen value, it may be quickly guessed. Meanwhile, it cannot protect heap buffer overflows.

Data Execution Prevention (DEP). DEP [4] prevents code-injection attacks by enforcing the stack/heap memory is non-executable, so the attacker cannot directly write shellcode to these memory pages. We can find DEP has no protection against code reuse attacks. Moreover, DEP is not compatible with programs that generate and modify code on-the-fly, such as Just-In-Time (JIT) compilers.

Address Space Layout Randomization (ASLR). The basic idea of ASLR [83] is to make it harder for attackers to precisely locate the reusable code. Specifically, program modules such as the executable file and dependent libraries are compiled to be position-independent and loaded into randomized addresses. However, ASLR can be bypassed by information disclosure and brute force attacks [74]. Some fine-grained code layout randomization approaches [38, 61, 88] are proposed, in which permutation on functions, instruction layout, basic blocks, and code transformation are implemented. However, they are vulnerable to JIT-ROP [79].

2.3 Control-Flow Integrity

Stronger than all the deployed defenses, Control-Flow Integrity (CFI) forces control-flow transfers in the program to follow the policy presented by the CFG.

Control-flow transfers can be either direct or indirect. Direct edges include sequential instruction execution and direct branching. For example, the transfer from a direct call instruction to its target function address is a direct control-flow transfer. Since targets of direct control transfers cannot be arbitrarily controlled by attackers, they are less of concern. Indirect transfers through indirect branch instructions including indirect calls, indirect jumps, and returns are more dangerous, because their targets may be arbitrarily controlled by attackers. To ensure CFI for indirect branches, they are checked before execution so that their targets are always legal.

In general, it can be classified into two categories: coarse-grained CFI and fine-grained ones.

2.3.1 Coarse-grained CFI

Having accurate and practical enforcement of CFI is known to be hard. First, it is generally difficult to accurately identify the target locations for all control transfers. Existing solutions typically apply a coarse-grained policy (e.g., to allow indirect calls to any functions) for Commercial Off-The-Shelf (COTS) software whose source code is unavailable. This kind of CFI marks the valid targets of indirect control transfers with unique identifiers (IDs) and then inserts ID-checks into the program before each indirect branch transfer. An indirect branch is allowed to jump to any destination with the same ID.

CFIMon [90] uses three IDs for all indirect branch transfers. The target of a return instruction can be any call-preceded basic blocks and the target of an indirect call can be any function. The valid targets for indirect jumps are obtained by making use of online training. It leverages Branch Trace Store (BTS) mechanism [37] supported by hardware to collect in-flight control transfers, and once the BTS buffer is nearly full, a monitor process will start to compare them with the valid targets to decide whether there exists an attack.

BinCFI [97] uses two IDs for all indirect branch transfers: one for ret and indirect jump instructions, another for indirect call instructions. All indirect branches are instrumented to jump to the corresponding address translation routine that determines the targets of the transfers, if one target cannot be found, it means there is a control-flow hijacking attack. However, whenever there is a control-flow transfer, the CFI checks are always executed. Our proposed approach *DynCFI* does not require to perform CFI check for each control-flow transfer due to the trace mechanism used in the dynamic code optimization platform DynamoRIO [12]. Specifically, the trace mechanism can avoid some indirect branch lookups by inlining a popular target of an indirect branch into a trace.

CCFIR [94] implements a 3-IDs approach, which extended the 2-IDs approach by further separating returns to sensitive and non-sensitive functions. All control-flow targets for indirect branches are collected and randomly allocated on a so-called springboard section, and indirect branches are only allowed to use control-flow targets contained in the springboard section. CCFIR can manage the indirect branch transfers better, and the targets of indirect branches are more restricted than other approaches. However, in the springboard section, there are other indirect branches, and memory disclosure can reveal the content of the entire springboard section, which can be leveraged by attackers.

BinCC [87] enforces a 4-IDs approach by dividing the binary code into several mutually exclusive code continents and further classifying each indirect transfer within a code continent as either an Intra-Continent transfer or an Inter-Continent transfer. Different continent transfers will have different valid targets. For example, the valid targets of Intra-Continent transfer are always inside their own code continent. For other Inter-Continent transfers, the policy is indirect call nodes can only reach all root nodes in all continents, indirect return nodes can only reach indirect call nodes, and indirect jump nodes can only reach all root and call nodes.

Since this kind of CFI approaches recover a coarse-grained CFG, they may be bypassed by some advanced code-reuse attacks [16, 36, 72]. A lot of fine-grained CFI approaches are proposed.

2.3.2 Fine-grained CFI

Most of these CFI approaches rely on the availability of source code. MCFI [57] propagates source-level information such as type information to the binary level as metadata, and gathers such metadata at program load time to build a precise CFG, which is consulted (or read) by the program to detect CFI violations. Specifically, when a code module is loaded during execution, the loading module's metadata is combined with the loaded module's metadata to compute a CFG for both modules, and then the old CFG will be replaced with the new CFG. πCFI [58] starts a program with an empty CFG and let the program itself lazily compute the CFG on the fly. The main idea behind this empty CFG approach is to affix edges on runtime prior to being used for branch instructions.

Forwarding CFI [84] protects binaries by inserting checks before all forward-edge control-flow transfers to check whether the function signatures (return type and the number of arguments) are correct. Cryptographically enforced CFI [51] enforces another form of fine-grained CFI by adding a message authentication code (MAC) that is computed with type information to control-flow elements, which prevents the usage of unintended control-flow transfers in the CFG.

Since the requirement of source code makes these approaches difficult to be deployed. CFI based on function signature matching at the binary level is proposed. There is two work focus on it called TypeArmor [85] and τCFI [55]. TypeArmor is the first work that uses the function signature on the number of arguments at the binary level to enforce CFI. It extracts the number of arguments both at the caller and callee sites by performing backward and forward analysis, and the target of the indirect call can only be functions that have matching signatures. τCFI is the follower of TypeArmor that tries to construct a more fine-grained CFG by combining the width of the argument registers as the additional function signature.

Both of them rely on the x86-64 calling convention that the first six arguments for integer are passed through registers (assuming System V ABI). However, the compiler may generate code that violates the calling convention. For example, modern compilers may not set the argument register explicitly at the caller site if it finds the argument value is already in the corresponding register. We systematically study how compiler would impact function signature recovery for TypeArmor [85] and τCFI [55] and find compiler optimizations have a great impact on function signature recovery. Such errors could lead to invalid function calls being allowed or, even worse, valid calls being inadvertently blocked. The proposed approach that makes use of deep learning can recover function signature more accurately compared to TypeArmor [85] and τCFI [55]. Moreover, it does not rely on the calling convention.

Both coarse-grained and fine-grained approaches usually have high performance overhead as for every indirect control-flow transfer they need to do CFI check. Moreover, they need to add CFI checks into the code section or consult read-only data structures. If these CFI checks and read-only data structures are compromised, these mitigation approaches can be bypassed easily. Our proposed system C^3 embeds the CFI policy into every machine instruction without relying on the assumption of keeping such metadata read-only. Furthermore, all of them do not insert CFI checks for unintended control-flow transfers, making them being bypassed by the exploit proposed by Conti et al. [22]. Such exploits would not work on C^3 as all instructions (intended or unintended) are encrypted.

2.4 Instruction-Set Randomization

Instruction-Set Randomization (ISR) was initially proposed to fight against code-injection attacks [42, 7, 65, 34]. It encrypts instructions and provides a unique instruction set to every program. Injected code would first be decrypted to a random byte sequence and result in illegal instructions executed. Recently, researchers look into using ISR to defend against code-reuse attacks (CRA). Scylla [82] encrypts every instruction in a basic block with respect to its predecessor to defend against

CRA that jumps to the middle of a basic block. However, it does not stop attacks that make use of the entire basic block to construct gadgets. Polyglot [78] combines ISR with fine-grained code randomization to defend against JIT-ROP [79]. It encrypts every basic block of instructions by XORing them with the starting address of the block. Since the encryption key is only derived from the address of the basic block and not tied to control transfers, CFI can be compromised with control transfers from invalid callers. C^3, on the other hand, is designed for enforcing CFI with encryption key tied to all valid control transfers. Invalid callers will result in the wrong decryption key generated and random code bytes obtained. SOFIA [31] uses ISR to enforce CFI for cyber-physical systems with instructions at a fixed length of 32-bit via an integrity check of instruction blocks where the Message Authentication Code (MAC) is encrypted. SOFIA requires access to source code of the program to be protected in order to rearrange the CFG so that every basic block has up to two callers. On the other hand, we propose a novel idea C^3 which uses secret sharing to support multiple (potentially more than two) callers.

2.5 Function Signature Recovery

Besides TypeArmor [85], liveness analysis and heuristic methods based on calling conventions and idioms were used to recover function signatures. EIWazeer et al. [33] apply liveness analysis to recover arguments, variables, and their types for x86 executables. It assumes all registers are arguments to every function and then traverse the call graph of the executable in post-order depth-first search traversal to check whether there is a “real” use of this register in the function. Only “real” used registers are considered as arguments. The argument that stored on the stack is recognized by making use of Value Set Analysis [6]. Points-to analysis is used to recover the type of an argument (variable) according to some type recovering rules.

TIE [45] infers variable types in binaries through formulating the usage of different data types. It first lifts the binary code to a binary analysis language called

BIL by using BAP [13], and then type information is inferred by solving type constraints. It depends on that some known prototype functions will be called (e.g., library functions), so that the constraint can be solved by making use of some rules. Caballero et al. [14] make use of dynamic liveness analysis to recover function arguments for execution traces. Since it is a dynamic analysis, it cannot guarantee the full coverage of unused arguments during an execution trace. Recently, Zeng et al. [93] propose to perform type inference based on debugging information generated by the compiler so that a high-precision CFG can be constructed to help CFI enforcement. Another direction is to make use of machine learning approaches to recover function signatures. For example, EKLAVYA [20] uses a three layers Recurrent Neural Network to learn the number and types of arguments from disassembled binary code. More details about EKLAVYA can be found in Section 3.2.1.

2.6 Non-control Data Attacks

Compared to control-flow hijacking attacks, Non-control attacks manipulate non-control data to alter a program's benign behavior without violating its control-flow integrity. This kind of non-control data attacks can cause significant damages, such as leakage of secret keys (HeartBleed) [1] and enabling untrusted code import [92]. Hu et al. [39] make use of data-flow stitching to systematically finds ways to join data flows in the program to generate data-oriented exploits. Specifically, it takes as input a vulnerable program with a memory error, an input that exploits that memory error, and a benign input that triggers the same execution path (two-dimensional data-flow graph (2D-DFG)), and uses backward and forward slicing to pinpoint data flow paths between inputs and pre-identified sensitive data.

In a follow up work, Hu et al. [40] propose data-oriented programming (DOP) which demonstrates that non-control data attacks resulting from a single memory error can be Turing-complete, and a large number of data-oriented gadgets can be found. These gadgets require to deliver operation result with memory and must execute in at least one legitimate control flow, and need not execute immediately one after another. By stitching these gadgets together using a dispatcher (e.g., loop), the attacker can perform arbitrary operations. It has shown that DOP can be used to change the permissions of read-only pages to bypass current CFI implementations by triggering `dlopen`'s internal gadgets to corrupt the read-only pages.

Chapter 3

Function Signature Recovery

In this chapter, we first introduce our work on studying how compiler optimizations impact the accuracy of function signature recovery on x86-64 platform, and then present the enhanced approach to recover function signature by making use of recurrent neural network.

3.1 When Function Signature Recovery Meets Compiler Optimization

In this section, we first theoretically analyze the possible ways in which compiler optimizations could impact the accuracy of two most recent approaches in function signature recovery for CFI, namely TypeArmor [85] and τCFI [55], and then experiment with a large number of applications including Binutils[1], LLVM test-suite[2], as well as C/C++ applications from Github to evaluate the extent to which such complications arise on real-world applications. We recover the ground truth of function signatures of 552 C and 792 C++ applications compiled with `gcc-8` and `clang-7` with optimization levels `-O0` to `-O3` and compare them with results of TypeArmor [85], τCFI [55], and Ghidra [35] in recovering the number of arguments

[1]https://www.gnu.org/software/binutils/
[2]https://llvm.org/docs/TestSuiteGuide.html

and argument types.

Results show that compiler optimizations have both positive and negative impacts on function signature recovery. First, optimizations make the identification of variadic functions more accurate as arguments are more likely to be moved to callee-saved registers than being moved onto the stack. At the same time, the elimination of redundant instructions due to optimization also simplifies the argument analysis at callers. However, compiler optimization could make identification of the number of arguments and the type inferencing at callees less accurate, because of the elimination of unused arguments and promotion/demotion of argument types.

In order to mitigate these inaccuracies, we propose our improved policies to recover the function signatures more accurately from optimized binaries. We evaluate our proposed policies with the same set of real-world applications and compare our accuracy with that of existing ones. Results show that, e.g., the likelihood of misidentifying variadic functions in C is reduced from 3.3% to 1.2%. Moreover, our policy can mitigate all issues caused by argument type demotion at callers and argument type promotion at callees.

3.1.1 Background and Unified Notation

On Linux x86-64, all arguments of a function are passed from the caller to the callee who is assumed to process every argument. Integer arguments are passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` in sequence, while `%XMM0 - %XMM7` are used to pass floating-point arguments [52]. Additional arguments are pushed onto the stack in reverse order. The return value is stored in `%rax` with potentially the higher 64 bits stored in `%rdx`. Floating-point return values are similarly stored in `%XMM0` and `%XMM1`. Both TypeArmor and τCFI adhere to these calling conventions and do not consider deviations from them.

Variadic functions (such as `printf` in the C library) are used to maximize flexibility in argument passing. These functions accept a variable number of arguments which do not necessarily have fixed types.

TypeArmor [85] and τCFI [55] reconstruct both callee and caller signatures by performing static binary analysis and then use this information to enforce Control-Flow Integrity between callees and callers with similar signatures. TypeArmor uses the number of arguments as the signature, while width (number of bits $p \in \{64, 32, 16, 8\}$) of the argument-storing registers is used by τCFI. Just like in existing approaches, we focus on function signature recovery for integer arguments and use $i \in [1, 6]$ to index the six argument registers. Here we introduce our unified notation to describe the CFI policies TypeArmor and τCFI employ as well as our improved policy (see Section 3.1.4).

Analysis of callees

Analysis of a callee function typically starts from the function entry and continues in a forward manner until the end of the function. Here, the analysis focuses on the *first* instruction involving a argument-passing register, which could have one of the following four possible states: $s^{EE} \in \{\dot{w}(), r\dot{w}(), r\dot{w}2s(), c\}$ (we use the dot above a state to denote that it's the analysis result of the *first* instruction involving the corresponding register).

Definition 1. *State $\dot{w}_i(p)$ if the first instruction involving register i is writing into the lower p bits of register i.*

Definition 2. *State $r\dot{w}_i(p)$ if the first instruction involving register i is reading the lower p bits of it and writing to anther register or a non-stack address.*

Definition 3. *State $r\dot{w}2s_i(p)$ if the first instruction involving register i is reading the lower p bits of it and writing to a stack address.*

Definition 4. *State c_i if register i is not involved in any instructions.*

Definition 5. *Argument register state vector observed at callee $P_{EE}^{OB} =< s_1^{EE}, s_2^{EE}, s_3^{EE}, s_4^{EE}, s_5^{EE}, s_6^{EE} >$ where $s_i^{EE} \in \{\dot{w}_i(), r\dot{w}_i(), r\dot{w}2s_i(), c_i\}$ for $i \in [1, 6]$.*

Definition 6. *$b2b_i$ is true if $s_i^{EE} = r\dot{w}2s_i()$ and $s_{i+1}^{EE} = r\dot{w}2s_{i+1}()$ and the corresponding instructions involving registers i and i +1 are back to back.*

Analysis of callers

Analysis of a caller function starts at the indirect call instruction and continues in a backward manner until it hits another function call instruction. This backward analysis follows the CFG and focuses on *all instructions* involving the argument-passing register instead of only the first instruction as in the analysis of callees.

Definition 7. *State $w_i(p)$ if there is an instruction writing to the lower p bits of register i.*

Definition 8. *State $\hat{w}_i$ if there is no instruction writing to register i.*

Definition 9. *Argument register state vector observed at caller $P_{ER}^{OB} =< s_1^{ER}, s_2^{ER}, s_3^{ER}, s_4^{ER}, s_5^{ER}, s_6^{ER} >$ where $s_i^{ER} \in \{w_i(), \hat{w}_i\}$ for $i \in [1, 6]$.*

TypeArmor's Policy on the Number of Arguments

At a callee, TypeArmor [85] performs a forward recursive analysis from the entry block to find out states of the six argument registers. If the state of the sixth argument register (`%r9`) is $r\dot{w}2s_6()$, TypeArmor concludes that this function is variadic and the number of arguments is the maximal i that makes $b2b_i$ false. If the state of `%r9` is not $r\dot{w}2s_6()$, the function is considered non-variadic and the number of arguments is the maximal i with state $r\dot{w}2s_i()$ or $r\dot{w}_i()$.

Definition 10. *The observed number of arguments at callee $|P_{EE}^{OB}|$ is:*

$$\begin{cases} \underset{i}{\operatorname{argmax}}(\neg b2b_i) & \textit{if } s_6^{EE} = r\dot{w}2s_6() \\ \max(\underset{i}{\operatorname{argmax}}(r\dot{w}2s_i()), \underset{i}{\operatorname{argmax}}(r\dot{w}_i())) & \textit{otherwise} \end{cases}$$

TypeArmor iterates over each indirect caller and performs a backward static analysis to detect the number of arguments prepared. If the states of all argument registers are $w()$, TypeArmor stops the analysis and considers that the caller prepares the maximum number of arguments. If some argument registers are neither $w()$ nor $\hat{w}$, TypeArmor performs a recursive backward analysis on incoming control flows. In cases where incoming control flows are via indirect calls and therefore backward analysis fails in identifying the caller function, TypeArmor assumes that the maximum number of arguments is prepared. It also assumes that the argument registers are always reset between two function calls, and therefore analysis is terminated when a return edge is encountered. In summary, the number of arguments at the caller is the minimal i with state $\hat{w}_i$ minus one.

Definition 11. *The observed number of arguments at caller $|P_{ER}^{OB}|$ is:*

$$\begin{cases} \underset{i}{\operatorname{argmin}}(s_i^{ER} = \hat{w}_i) - 1 & \textit{if } \exists \hat{w}_i \in P_{ER}^{OB} \\ 6 & \textit{otherwise} \end{cases}$$

Since there could be overestimation at callers and underestimation at callees, TypeArmor allows caller A to call callee B if and only if $|P_{ER-A}^{OB}| \geq |P_{EE-B}^{OB}|$.

τCFI's Policy on the Width of Arguments

τCFI [55] is the follower of TypeArmor that constructs a more fine-grained CFG by additionally considering the widths of argument registers as function signatures. It analyzes the number of bits of argument registers that are read or written to at callees and callers, respectively. We use $|s^{EE}|$ and $|s^{ER}|$ to represent the with of arguments at callees and callers, respectively. For example, if $P_{EE}^{OB} =< \dot{w}_1(), \dot{w}_2(), \dot{w}_3(), \dot{w}_4(), r\dot{w}_5(64), r\dot{w}_6(64) >$, then $|s_1^{EE}| = |s_2^{EE}| = |s_3^{EE}| = |s_4^{EE}| = 0$ and $|s_5^{EE}| = |s_6^{EE}| = 64$.

Since the analysis could cause overestimation at callers and underestimation at callees, the CFI policy of τCFI is: caller A can transfer control flow to callee B if and only if: $\forall i \in [1, |P_{ER}^{OB}|], |s_i^{ER}| >= |s_i^{EE}|$.

We also denote the ground truth for the states of argument registers at callees and callers as P_{EE}^{GT} and P_{ER}^{GT}, respectively. $|s^{EE,GT}|$ and $|s^{ER,GT}|$ are used to denote the ground truth on the width of arguments.

3.1.2 Eight Ways in which Compiler Optimization Impacts Function Signature Recovery

In this section, we present our analysis in binary optimization strategies and how they impact the accuracy of function signature recovery. Specifically, we study the source code of compilers (`gcc-8` and `clang-7`), paying special attention to the mechanism in which arguments are passed from callers to callees under different optimization flags (`-O0`, `-O1`, `-O2`, `-O3`). We also consult the Intel instruction manual [41] on how each instruction could affect function signatures. Finally, we compile the following eight scenarios in which compiler optimization could impact function signature recovery by the two most recent work, namely TypeArmor and τCFI.

Misidentifying variadic functions

As outlined in Section 3.1.1, TypeArmor uses $r\dot{w}2s_6()$ as the sole indicator of a variadic function. Interestingly, such a policy tends to introduce more errors in unoptimized binaries in which all arguments are moved onto the stack and any normal function with more than five arguments will be misidentified as variadic. We denote this complication as ***Nor2Var***. On the other hand, optimized binaries tend to move arguments to callee-saved registers, which reduces the chances of such errors. That said, normal functions in optimized binaries may still use the stack for argument passing if the compiler determines that the argument will be reused after the call.

Listing 3.1a shows a function compiled with `clang -O2`. Since $s_6^{EE} = r\dot{w}2s_6()$, $b2b_5$ is true and $b2b_4$ is false. TypeArmor determines that `coff_write_symbol` is a variadic function with 4 arguments. However, $|P_{EE-0x471a60}^{GT}| = 7$ as shown at Line 1.

```
static bfd_boolean coff_write_symbol (*,*,*,*,*,*,*)
0000000000471a60 <coff_write_symbol>:
......
471a6e:    mov    %r9,0x40(%rsp)
471a73:    mov    %r8,0x10(%rsp)
471a78:    mov    %rcx,%r15
471a7b:    mov    %rdx,%r14
471a7e:    mov    %rsi,%rbp
471a81:    mov    %rdi,%r12
......
471c4f:    mov     0x40(%rsp),%rbx
```

a: Normal function misidentified as variadic

```
void bfd_set_error (bfd_error_type error_tag, ...) {
  bfd_error = error_tag;
  if (error_tag == bfd_error_on_input) {
    va_list ap;
    va_start (ap, error_tag);
    input_bfd = va_arg (ap, bfd *);
    input_error = (bfd_error_type)va_arg(ap,int);
    ......
  }
}
00000000000328c0 <bfd_set_error>:
......
328c4:   mov    %edi,0x300186(%rip)
......
328da:   cmp    $0x14,%edi
328dd:   mov    %rsi,0x28(%rsp)
328e2:   mov    %rdx,0x30(%rsp)
328e7:   je     32900
```

b: Variadic function misidentified as normal

```
char *concat_copy(char *dst, const char *first, ...)
00000000000dea00 <concat_copy>:
......
dea25:     test %rsi,%rsi
dea28:     mov %rdx,0x30(%rsp)
dea2d:     mov %rcx,0x38(%rsp)
dea32:     mov %r8,0x40(%rsp)
dea37:     mov %rax,0x8(%rsp)
dea3c:     lea 0x20(%rsp),%rax
dea41:     mov %r9,0x48(%rsp)
```

c: Number of default arguments overestimated

Listing 3.1: Examples of variadic function misidentification

Another complication arises when a variadic function does not use some of the variadic arguments. An optimized binary will not explicitly read these arguments, which will cause the variadic function to be misidentified as normal (denoted as ***Var2Nor***). Note that this does not affect binaries compiled by `clang` since `clang` always explicitly reads all variadic arguments.

Listing 3.1b shows a variadic function `bfd_set_error` compiled by `gcc -O2`. As shown at Line 6 – 7, only the first two variadic arguments are used by this function, and therefore `gcc` only moves `%rsi` and `%rdx` onto the stack (Line 16 – 17). Current approaches would find that $P^{OB}_{EE-0x328c0} =< r\dot{w}_1(32), r\dot{w}2s_2(64), r\dot{w}2s_3(64), c_4, c_5, c_6 >$ and determine that $|P^{OB}_{EE-0x328c0}| = 3$ since `%r9` is not moved onto the stack. However, $|P^{GT}_{EE-0x328c0}| = 1$ as shown at Line 1.

Moreover, instructions that move the variadic arguments onto the stack in an optimized binary may not be back to back, which results in $b2b$ being unreliable in determining the number of arguments — an overestimation (denoted as ***VarOver***). Listing 3.1c shows the variadic function `concat_copy` compiled by `gcc -O2`. TypeArmor and τCFI find $b2b_5$ to be false and determine that it is a variadic function with 5 default arguments, but the ground truth is it has only 2 default arguments as shown at Line 1.

Missing argument-reading instructions

When optimization is enabled, there may not be explicit reading of an argument if the function does not use it, leading the corresponding state of the argument to be c. We denote this complication as ***Unread***. As shown in Listing 3.2, since the first and third arguments of `jpeg_free_large` (compiled by `clang -O2`) are not used, TypeArmor and τCFI determine that $P^{OB}_{EE-0x41b6b0} =< \dot{w}_1(64), r\dot{w}_2(64), c_3, c_4, c_5, c_6 >$. Note that compilers always set the argument registers at callers even if they are not used by the callee; see Line 11 – 12.

```
GLOBAL(void) jpeg_free_large (j_common_ptr cinfo, void FAR
    * object, size_t sizeofobject) {
  free(object);
}
000000000041b6b0 <jpeg_free_large>:
41b6b0: mov    %rsi,%rdi
41b6b3: jmpq    400950 <free@plt>

caller site:
41b5a0: mov    0x70(%r14,%r15,8),%rsi
......
41b5d3: mov    %r12,%rdi
41b5d6: mov    %rbp,%rdx
41b5d9: callq   41b6b0 <jpeg_free_large>
```

Listing 3.2: Not reading argument registers

```
long test(long a, long b)
00000000004006a0 <test>:
......
4006ae: callq   400490 <lldiv@plt>
4006b3: mov    %rbx,%rdi
4006b6: mov    %rdx,%rsi
4006b9: callq   *0x200db1(%rip)    # 601470 <fptr3>
4006bf: mov    %rax,%rbx
4006c2: callq   *0x2009a0(%rip)    # 601068 <fptr4>
```

Listing 3.3: Misidentifying %rdx as an argument

Misidentifying %rdx as an argument

Some registers have special usage in addition to passing arguments. For example, the third argument register `%rdx` can also be used to store return values when the size of the return value is larger than 64 bits. When there is a read operation on it, current approaches do not distinguish reading an argument from reading the higher 64 bits of a return value. It could then result in an overestimation on the number of arguments. This complication is denoted as ***rdx***.

As shown in Listing 3.3, TypeArmor and τCFI determine that $P^{OB}_{EE-0x4006a0} =< r\dot{w}_1(64), \dot{w}_2(32), r\dot{w}_3(64), c_4, c_5, c_6 >$, and that it is a normal function with 3 arguments. However, $|P^{GT}_{EE-0x4006a0}| = 2$ and the reading of `%rdx` is for the higher 64 bits of the return value of function `lldiv`.

```
1  typedef unsigned int JDIMENSION;
2  void process_data_crank_post(j_decompress_ptr cinfo,
      JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
      JDIMENSION out_rows_avail) {
3    (*cinfo->post->post_process_data) (cinfo, NULL, NULL, 0,
       output_buf, out_row_ctr, out_rows_avail);
4  }
5  00000000000165c0 <process_data_crank_post>:
6  165c0:   sub     $0x10,%rsp
7  165c4:   mov     0x228(%rdi),%rax
8  165cb:   mov     %rsi,%r8
9  165ce:   mov     %rdx,%r9
10 165d1:   push    %rcx
11 165d2:   xor     %edx,%edx
12 165d4:   xor     %ecx,%ecx
13 165d6:   xor     %esi,%esi
14 165d8:   callq   *0x8(%rax)
```

Listing 3.4: Promoted argument pushed onto the stack

Argument (width) promotion

Some instructions may only work on 64-bit registers or memory, and optimization may prefer using 64-bit registers since using 32-bit registers would result in longer instructions. For example, the compiler uses `push` to pass arguments to callees (via the stack) when the flag "-mpush-arg" is enabled (e.g., when it is the 7th argument). However, `push` only allows 64-bit registers as operands, which leads to argument (width) promotion (denoted as ***Push***). Line 1 – 4 of Listing 3.4 shows that the fourth argument `out_row_avail`, whose type is unsigned int, is passed as the 7th argument at Line 3, and is pushed onto the stack at Line 10 (resulting in $rw_4(64)$ instead of $rw_4(32)$).

Another complication is due to the default width of operands of certain instructions, e.g., `lea` [41]. Compilers prefer reading a 64-bit register even if the width of the argument is 32 bits, since reading a 32-bit register requires a prefix `67H` (denoted as ***lea***). Listing 3.5 shows an example of it, the state of the second argument is $rw_2(64)$; however, the ground truth is a 32-bit argument (unsigned int).

```
bfd_check_overflow (enum complain_overflow how,
unsigned int bitsize , unsigned int rightshift , unsigned int
    addrsize , bfd_vma relocation)

000000000048ca60 <bfd_check_overflow >:
48ca60: mov     %ecx,%eax
48ca62: mov     %edx,%r9d
48ca65: lea     -0x1(%rsi),%ecx
48ca68: mov     $0xfffffffffffffffe,%rdx
```

Listing 3.5: Promoted operand of instruction lea

Missing argument-writing instructions

Similar to missing argument reading instructions at callees as discussed above, compiler optimization may decide not to set or reset the value of a register explicitly at callers.

- Higher 64 bits of the return value used as the third argument (denoted as ***Ret***). `%rdx` is used to store the higher 64 bits of the return value. If the compiler finds that a function uses this value as the third argument, it will not explicitly reset `%rdx` again.
- Uninitialized variable as an argument (denoted as ***Uninit***). `clang` generates `undef` values for uninitialized variables and do not explicitly set these arguments [46, 54]. On the other hand, `gcc` initializes them to zero[3].
- Indirect calls in wrapper functions (denoted as ***Wrapper***). Indirect callers may not reset argument registers when their values are already in the corresponding registers especially for inlined functions.
- Argument values not modified between two calls (denoted as ***Unmodified***). `gcc-7` and above eliminates writing across functions when the argument register is set to the same value for two consecutive callers.

All the above except ***Wrapper*** leads to $\hat{w}$ and results in underestimation on the number of arguments. Listing 3.6a presents the higher 64-bit return value

[3]https://github.com/gcc-mirror/gcc/blob/master/gcc/init-regs.c

and an uninitialized variable are used as arguments. The state vectors for the two indirect calls are $P^{OB}_{ER-0x4006b9} =< w_1(64), w_2(64), \hat{w}_3, \hat{w}_4, \hat{w}_5, \hat{w}_6 >$ and $P^{OB}_{ER-0x4006c2} =< \hat{w}_1, \hat{w}_2, \hat{w}_3, \hat{w}_4, \hat{w}_5, \hat{w}_6 >$, respectively, which lead to a finding of $|P^{OB}_{ER-0x4006b9}| = 2$ and $|P^{OB}_{ER-0x4006c2}| = 0$. However, by observing the source code at Line 6 – 7, we realize that $|P^{GT}_{ER-0x4006b9}| = 3$ and $|P^{GT}_{ER-0x4006c2}| = 2$. Listing 3.6b shows indirect calls in a wrapper function. Since there is no direct caller for function `bfd_elf64_swap_dyn_in`, TypeArmor and τCFI determine that $P^{OB}_{ER-0x416845} =< w_1(64), w_2(64), w_3(64), w_4(64), w_5(64), w_6(64) >$, which results in an overestimation on the number of arguments while $|P^{GT}_{ER-0x416845}| = 1$. Listing 3.6c shows that $P^{GT}_{ER-0x1aae2c} =< w_1(64), w_2(64), w_3(64), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$. However, $P^{OB}_{ER-0x1aae2c} =< w_1(64), \hat{w}_2, w_3(64), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$ and $|P^{OB}_{ER-0x1aae2c}| = 1$ since the value of `%rsi` is not changed by the function at `0x1a95f0`, and the compiler does not reset it explicitly.

Registers storing temporary values

Since all argument registers are general-purpose registers, they could also be used as scratch registers to store temporary values, which could result in an overestimation on the number of arguments (denoted as ***Temp***). Listing 3.7a shows an example (compiled with `clang -O0`) with $P^{OB}_{ER-0x416015} =< w_1(64), w_2(64), w_3(64), w_4(64), \hat{w}_5, \hat{w}_6 >$ and $|P^{OB}_{ER-0x416015}| = 4$. However, according to the ground truth at Line 7, we can observe that $|P^{GT}_{ER-0x416015}| = 3$ and the write operation on `%rcx` is to store a temporary value. Note that compiler optimization can remove many redundant instructions that are used to store temporary values; and so it has a positive impact on this case; see the optimized binary in Listing 3.7b where $P^{OB}_{ER-0x438891} =< w_1(64), w_2(64), w_3(64), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$ and $|P^{OB}_{ER-0x438891}| = |P^{GT}_{ER-0x438891}| = 3$.

```
long test2(long a, long b){
  //mesg and err are not initialized
  char *mesg,*err;
  lldiv_t res;
  res = lldiv (31558149LL,3600LL);
  long r1 = (*fptr3)(a, res.quot, res.rem);
  (*fptr4)(mesg,err);
  printf("%s\n", buffer);
  return r1;
}
00000000004006a0 <test2>:
......
4006ae: callq  400490 <lldiv@plt>
4006b3: mov    %rbx,%rdi
4006b6: mov    %rax,%rsi
4006b9: callq  *0x200db1(%rip) # 601470 <fptr3>
4006bf: mov    %rax,%rbx
4006c2: callq  *0x2009a0(%rip) # 601068 <fptr4>
```

a: Higher 64-bit return value and uninitialized variable used as arguments

```
0000000000416830 <bfd_elf64_swap_dyn_in>:
416830: push   %r15
......
416835: mov    %rdx,%r14
416838: mov    %rsi,%r15
41683b: mov    %rdi,%rbx
41683e: mov    0x8(%rdi),%rax
416842: mov    %rsi,%rdi
416845: callq  *0x68(%rax)
```

b: An indirect call in a wrapper function

```
1aae0a: mov    0xb38(%r13,%r14,1),%rdi
1aae12: mov    %rbp,%rsi
1aae15: callq  1a95f0
1aae1a: mov    (%rsp),%rax
1aae1e: lea    (%rax,%r14,1),%rdx
1aae22: mov    (%rbx),%rax
1aae25: mov    0xb8(%rax),%rdi
#call funcs->create( cffsize->face->memory, &priv, &
    internal->subfonts[i - 1] )
1aae2c: callq  *(%r12)
```

c: Arguments not modified between two calls

Listing 3.6: Missing argument-writing instructions

```
......
460ffc:  mov    -0x18(%rbp),%rdi
461000:  mov    -0xe8(%rbp),%rsi
461007:  mov    -0xf0(%rbp),%rcx
46100e:  add    $0x10,%rcx
461012:  mov    %rcx,%rdx
#(*bed->elf_backend_reloc_type_class)(info, o, s->rela);
461015:  callq  *%rax
```

a: Assembly compiled with clang -O0.

```
......
438881:  mov    0x30(%rsp),%rdi
438886:  mov    %rbx,%rsi
438889:  mov    %rbp,%rdx
43888c:  mov    0x10(%rsp),%rax
#(*bed->elf_backend_reloc_type_class)(info, o, s->rela);
438891:  callq  *0x208(%rax)
```

b: Assembly compiled with clang -O2.

Listing 3.7: Registers to store temporary values

Argument (width) demotion

To the opposite of argument promotion at callees, compilers may use a smaller-sized register (32-bit), since a 64-bit register may need a REX prefix [41] which increases the code size and affects the I-cache footprint. This applies to cases where

- Arguments are constants whose sizes are up to 32 bits (denoted as ***Imm***);
- Arguments are pointers pointing to .rodata, .bss, and .text sections (denoted as ***Pointer***); and
- Arguments are NULL pointers (denoted as ***Null***).

Listing 3.8a shows an example for these cases compiled by `clang -O2`. The ground truth at Line 4 shows $P^{GT}_{ER-0x546593} =< w_1(64), w_2(64), w_3(64), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$, while TypeArmor and τCFI determine that $P^{OB}_{ER-0x546593} =< w_1(64), w_2(32), w_3(32), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$ since the second argument (`0x8a01b0`) is a pointer pointing to the .rodata section, and the third argument (`0x2000`) is a 32-bit constant. The example with a NULL pointer being

```
1 546586:mov $0x8a01b0,%esi
2 54658b:mov $0x2000,%edx
3 546590:mov %r14,%rdi
4 #(*git_hash_update_fn)(*, *, size_t len);
5 546593: callq *0x28(%rax)
```

a: A constant and a pointer as arguments

```
1 57f50e:test %rbp,%rbp
2 57f511:je 57f531
3 57f513:mov 0x333a46(%rip),%rdi
4 57f51a:xor %esi,%esi
5 #(*advertise)(*r, *);
6 57f51c: callq *0x8(%rbp)
```

b: A NULL pointer as an argument

Listing 3.8: Argument width demotion

an argument is shown in Listing 3.8b. According to the ground truth at Line 5, the second argument should be a pointer; but a NULL pointer is passed at the caller, and the compiler uses `xor` to prepare for it.

Argument (width) promotion at both callees and callers (*Prom*)

There are other argument (width) promotions at both callees and callers that would not result in inaccuracies in matching function callees with callers since the argument promotion happens in a matching manner. This refers to promotions of types smaller than the native type of the target platform's Arithmetic Logic Unit (ALU) to make arithmetic and logical operations possible or more efficient. C and C++ perform such promotions for objects of boolean, character, wide character, enumeration, and short integer types. As shown in Listing 3.9, the type of the third argument is unsigned char (8-bits) as shown at Line 1, but the analysis engine would determine its state being $r\dot{w}_3(32)$ due to the promotion performed by the compiler.

Table 3.1 presents a summary on the complications at both callees and callers with the last column indicating the consequences.

Table 3.1: Summary of complications introduced by compiler optimization

Site	Category	Complication	Impact
Callee	Misidentifying variadic functions	Normal to variadic (***Nor2Var***)	$\lvert P_{EE}^{OB}\rvert < \lvert P_{EE}^{GT}\rvert$
		Variadic to Normal (***Var2Nor***)	$\lvert P_{EE}^{OB}\rvert > \lvert P_{EE}^{GT}\rvert$
		Back-to-back condition unreliable (***VarOver***)	$\lvert P_{EE}^{OB}\rvert > \lvert P_{EE}^{GT}\rvert$
	Missing argument reading instructions	Arguments are not used by a function (***Unread***)	$\lvert P_{EE}^{OB}\rvert < \lvert P_{EE}^{GT}\rvert$ $\lvert s_i^{EE}\rvert < \lvert s_i^{EE,GT}\rvert$
	Misidentifying %rdx as an argument	Reading the higher 64 bits of a return value (***rdx***)	$\lvert P_{EE}^{OB}\rvert > \lvert P_{EE}^{GT}\rvert$
	Argument (width) promotion	Arguments are pushed onto the stack (***Push***)	$\lvert s_i^{EE}\rvert > \lvert s_i^{EE,GT}\rvert$
		Default width of the operand of certain instructions is 64-bit (***lea***)	$\lvert s_i^{EE}\rvert > \lvert s_i^{EE,GT}\rvert$
Caller	Missing argument writing instructions	Higher 64 bits of a return value as the third argument (***Ret***)	$\lvert P_{ER}^{OB}\rvert < \lvert P_{ER}^{GT}\rvert$
		Uninitialized variables as arguments (***Uninit***)	$\lvert P_{ER}^{OB}\rvert < \lvert P_{ER}^{GT}\rvert$
		Indirect calls in wrapper functions (***Wrapper***)	$\lvert P_{ER}^{OB}\rvert > \lvert P_{ER}^{GT}\rvert$
		Argument values not modified between two calls (***Unmodified***)	$\lvert P_{ER}^{OB}\rvert < \lvert P_{ER}^{GT}\rvert$
	Registers storing temporary values	Argument registers are used to store temporary values (***Temp***)	$\lvert P_{ER}^{OB}\rvert > \lvert P_{ER}^{GT}\rvert$
	Argument (width) demotion	Argumets are constant whose sizes are up to 32-bit (***Imm***)	$\lvert s_i^{ER}\rvert < \lvert s_i^{ER,GT}\rvert$
		Argument are pointers pointing to data and text sections (***Pointer***)	$\lvert s_i^{ER}\rvert < \lvert s_i^{ER,GT}\rvert$
		Arguments are NULL pointers (***Null***)	$\lvert s_i^{ER}\rvert < \lvert s_i^{ER,GT}\rvert$
Both	Small integral type promotion	Small integral types are promoted to native types (***Prom***)	$\lvert s_i^{EE}\rvert > \lvert s_i^{EE,GT}\rvert$ $\lvert s_i^{ER}\rvert > \lvert s_i^{ER,GT}\rvert$

```
 static bfd_boolean add_line_info (struct
line_info_table *table, bfd_vma address, unsigned char
op_index, char *filename,     unsigned int line,
unsigned int column, unsigned int discriminator, int
end_sequence)

 000000000044c2d0 <add_line_info>:
 44c2d0: push   %rbp
 ......
 44c2e7: mov    %edx,%r12d
 44c2ea: mov    %rsi,%r13
 44c2ed: mov    %rdi,%rax
 44c2f0: mov    (%rdi),%rdi
 44c2f3: mov    %rax,0x8(%rsp)
 44c2f8: mov    0x30(%rax),%rax
 44c2fc: mov    %rax,0x10(%rsp)
 44c301: mov    $0x28,%esi
 44c306: callq  408a80 <bfd_alloc>
```

Listing 3.9: Promotion of small integral types

3.1.3 Experimental Results of the Eight Complications on Real-World Programs

Section 3.1.2 details our theoretical analysis by analyzing compiler optimization strategies. In this section, we test how the eight complications identified in Section 3.1.2 present themselves in real-world programs. Specifically, we use a test suite of programs comprising of 552 C and 792 C++ applications compiled with `gcc-8` and `clang-7` with optimization levels from `-O0` to `-O3` for x86-64, and compare analysis results of TypeArmor and τCFI with ground truths extracted. Since the source code of τCFI is not released, we implement it ourselves according to the description of the paper [55].

In addition to TypeArmor and τCFI which recover function signatures for the specific purpose of Control-Flow Integrity, we also include a well-known binary analysis framework, Ghidra [35] v9.1.1, into our experiments since it also performs function signature recovery for reverse engineering purposes. Besides its general-purpose nature which leads to less emphasis on precision of the function signature recovery, our preliminary analysis on its source code reveals the following distinctions when Ghidra is compared to TypeArmor and τCFI in their mechanisms of function signature recovery:

- Only functions with symbol information are correctly identified as variadic, while those without symbol information are simply assumed to be non-variadic;
- Only instructions immediately prior to (without control-flow transfers) a call instruction are considered potentially preparing for function arguments;
- Forward and backward analysis are constrained within the scope of a single function; and
- Width for each argument at callers is always 64 bits.

With this preliminary understanding, we expect Ghidra to perform less accurately compared to TypeArmor and τCFI in recovering function signatures.

Our test suite is composed of Binutils-2.26, LLVM test-suite, and C/C++ applications from Github. This composition ensures that (1) it contains a wide variety of realistic C and C++ binaries with sizes ranging from 0.07MB to slightly more than 100MB (see Table 3.2 for details of sizes of the binary executables); (2) it contains binaries used in the evaluation of previous work, making it possible to compare our results with the literature; (3) it includes real-world applications downloaded from Github which contain complex corner cases which “testbed” applications may not have (see Table 3.3 for details of the Github applications we choose — mainly those with many “stars”).

Ground Truth and Statistics on the Ground Truth

Our objective of the experiments is to compare results from TypeArmor, τCFI, and Ghidra with ground truths to see how the complications identified in Section 3.1.2 present themselves in real-world applications. Here we first briefly explain how we obtain the ground truth in an automatic manner.

Table 3.2: Sizes of the binary executables in our test suite

Language	Opt	Size (MB)					
		clang			gcc		
		min	median	max	min	median	max
C	O0	0.07	0.69	44.75	0.08	0.68	44.72
	O1	0.07	0.71	45.61	0.12	0.98	50.52
	O2	0.08	0.84	50.09	0.11	1.02	51.79
	O3	0.08	0.84	48.95	0.13	1.55	54.30
C++	O0	0.11	7.51	65.77	0.12	14.60	73.22
	O1	0.11	7.22	68.82	0.17	10.32	99.95 9
	O2	0.13	6.31	65.70	0.18	16.96	105.50
	O3	0.13	6.15	66.79	0.19	17.12	109.83

Table 3.3: Github applications in our test suite

App	Language	description
git	C	Distributed version control system
darknet	C	An open source neural network framework
netdata	C	A real-time performance monitoring
redis	C	An in-memory database
sqlite	C	SQL database engine
vim	C	UNIX text editor
gnupg	C	Complete implementation of the OpenPGP standard
openssl	C	TLS/SSL and crypto library
mupdf	C & C++	A lightweight PDF, XPS, and E-book viewer
vorbis	C	A general purpose audio and music encoding format
aria2c	C++	A lightweight multi-protocol download utility
cppcheck	C++	Static analysis of C/C++ code
hpx	C++	C++ Standard Library for Parallelism and Concurrency
xpdf	C++	A PDF viewer and toolkit

We base our ground truth on information collected by an LLVM [44] pass and on DWARF v4 debugging information [21] which is the default setting for `gcc` and `clang`. We use LLVM to collect source-level information, including the number and types of arguments for each function and indirect callers when the arguments are integers (using LLVM API `isIntegerTy(N)`) and pointers (using LLVM APIs `isPointerTy()` and `isFunctionTy()`[4]). We also record the source line numbers of functions and indirect callers. We then compile the test applications with DWARF information and link the source-level line numbers with binary-level addresses using the DWARF line number table.

We implement the above with more than 500 lines of C++ code and more than 2,000 lines of python code. The result is a ground truth file for each binary in the

[4]We also check whether a struct argument has the attribute ByVal since clang will copy it onto the stack while considering it as a pointer.

Table 3.4: Number of arguments of functions in our test suite

Language	Opt	Number of Arguments (%)							# variadic
		0	1	2	3	4	5	6	
C	O0	6.92	29.35	29.73	17.46	7.47	4.33	1.77	8.45
	O1	6.01	28.64	29.87	17.32	7.73	4.71	1.95	
	O2	6.78	28.05	27.85	18.11	8.22	4.92	1.88	
	O3	5.99	26.52	29.04	18.20	8.55	5.17	2.11	
C++	O0	4.31	47.84	26.78	12.97	3.64	2.47	0.64	2.43
	O1	4.44	46.06	27.76	13.34	3.80	2.54	0.67	
	O2	3.09	45.27	20.77	12.58	7.09	5.48	1.87	
	O3	3.13	45.84	20.88	12.45	6.95	5.09	1.86	

test suite. With the ground truth collected, we perform statistical tests on our test suite to ensure that applications included could potentially present all variety of function signatures. Specifically, we count the number of arguments (ground truth) of all functions and make sure that there are sufficient numbers of functions with the number of arguments from 0 to 6; see Table 3.4 for details. We observe that there are more functions with between 1 and 3 arguments, and that C programs are more likely to have variadic functions. We also check the (ground truth) argument types for each function (see Table 3.5, which shows the percentage of functions having a specific type as its arguments). It appears that pointers are heavily used as function arguments, especially for C++ applications. This may imply that C++ applications are less likely to present complications on argument width demotion or promotion.

Metric Used and Overall Results

Since applications may have different numbers of functions and functions may have different numbers of indirect callers, we do not directly calculate the geometric mean as in TypeArmor [85] and τCFI [55]. Instead, we calculate the geometric mean of the *likelihood* that the callees and indirect callers present a complication in their function signature recognition. Specifically, we calculate the likelihood that the complications discussed in Section 3.1.2 cause under- and overestimation on the recovered function signatures. For example, application `addr2line` compiled with `clang -O0` has 2,019 normal functions among which 101 are misidentified

Table 3.5: Argument types of functions in our test suite

Type	Opt	Arg for C (%)						Arg for C++ (%)					
		1st	2nd	3rd	4th	5th	6th	1st	2nd	3rd	4th	5th	6th
8-bits	O0	0.19	0.37	0.12	0.26	0.34	0.35	0.02	0.32	1.52	0.27	0.83	0.94
	O1	0.14	0.31	0.23	0.29	0.54	0.56	0.08	0.56	1.60	0.38	0.43	0.55
	O2	0.11	0.24	0.30	0.23	0.75	0.78	0.02	0.31	1.48	0.26	0.79	0.92
	O3	0.10	0.25	0.26	0.29	0.60	0.61	0.02	0.32	1.52	0.27	0.83	0.94
16-bits	O0	0.09	0.19	0.13	0.14	0.17	0.29	-	0.04	0.22	0.24	0.44	0.50
	O1	0.11	0.24	0.17	0.12	0.13	0.27	0.04	0.08	0.11	0.22	0.39	0.31
	O2	0.10	0.21	0.10	0.16	0.14	0.33	-	0.04	0.22	0.23	0.42	0.49
	O3	0.11	0.24	0.15	0.13	0.14	0.29	-	0.04	0.22	0.24	0.44	0.50
32-bits	O0	9.38	19.58	25.33	29.65	33.50	28.79	0.82	9.66	19.57	26.02	29.27	17.36
	O1	8.55	19.75	25.29	30.31	37.60	32.97	0.55	4.60	15.19	15.57	21.03	16.80
	O2	8.31	18.41	24.21	28.38	32.71	25.61	0.81	9.44	19.10	24.89	27.83	16.98
	O3	7.48	19.23	24.91	30.48	38.84	33.33	0.82	9.66	19.57	26.02	29.27	17.36
64-bits	O0	2.31	7.25	11.36	10.37	10.85	10.00	0.14	5.13	10.82	17.14	13.03	10.47
	O1	1.97	6.17	10.93	9.18	9.01	7.74	0.83	7.54	14.87	11.21	7.37	6.29
	O2	2.24	6.99	12.19	11.08	10.77	9.83	0.14	4.96	10.92	17.21	13.04	10.31
	O3	1.94	5.95	10.65	9.31	8.69	7.57	0.14	5.13	10.82	17.14	13.03	10.47
ptr	O0	88.02	72.63	62.61	59.43	54.99	60.44	98.03	84.45	67.56	54.70	56.15	70.21
	O1	89.22	73.40	62.94	59.84	52.48	58.22	98.28	86.99	67.61	70.73	70.20	75.77
	O2	89.24	73.96	62.77	59.79	55.45	63.39	98.06	84.87	67.99	55.85	57.66	70.80
	O3	90.36	74.14	63.58	59.48	51.48	57.95	98.03	84.45	67.56	54.70	56.15	70.21

as variadic and the identified number of arguments is underestimated. We first calculate the likelihood that a function is misidentified in this application $(101/2019)$, and then use this number to compute the geometric mean for all applications in our test suite; see Figure 3.1[5] and Figure 3.2[6].

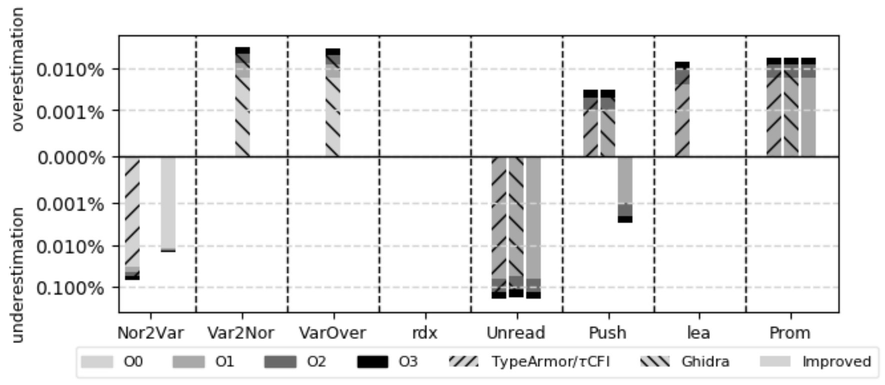
We discuss the detailed findings in the following sections. Note that complication case ***Unmodified*** only appears in one application (`mupdf`[7] compiled with `gcc`) and that ***Uninit*** and ***Ret*** do not appear at all in our test suite. We stress that this does not indicate insufficiency in our experiment, but rather the complications identified in our theoretical analysis (Section 3.1.2) do not necessarily present themselves in real-world programs.

***Unread*:** This is by far the biggest contributor to misidentification of function sig-

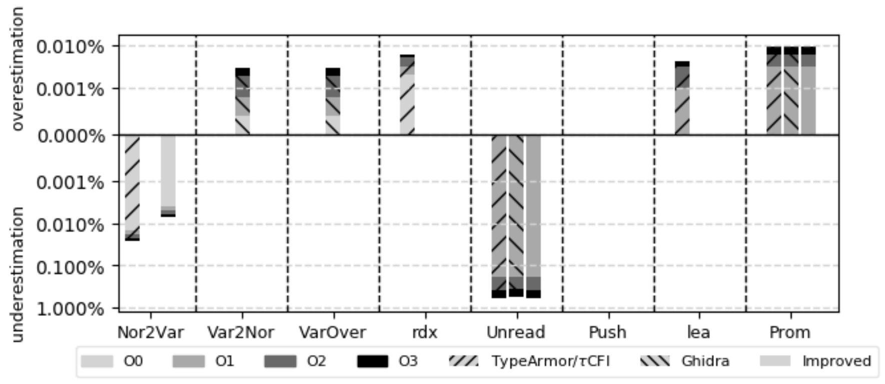
[5]Likelihood is calculated against the number of normal functions for ***Nor2Var***, against the number of variadic functions for ***Var2Nor*** and ***VarOver***, against the total number of functions for ***rdx***, ***Unread***, ***Push***, ***lea*** and ***Prom***. See Table 3.6 for the number of various types of functions in our test suite.

[6]Likelihood is calculated against the total number of indirect calls. See Table 3.6 for the number of indirect calls in our test suite.

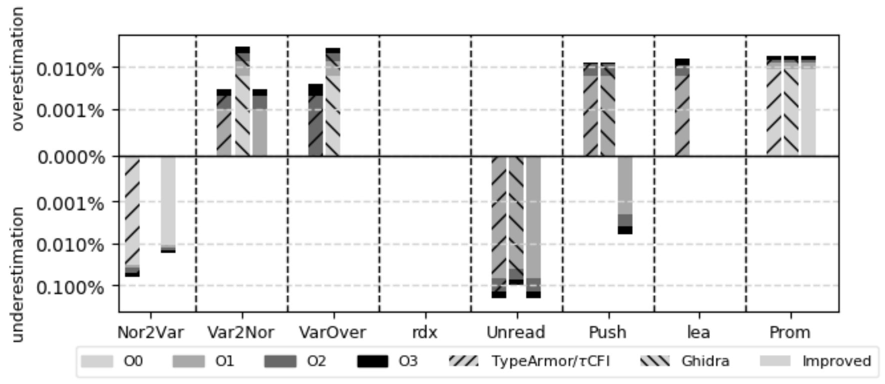
[7]https://mupdf.com/



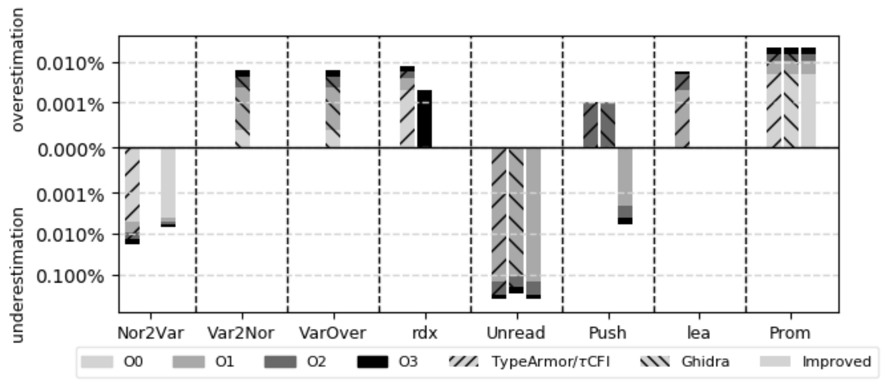
(a) C applications compiled by Clang



(b) C++ applications compiled by Clang

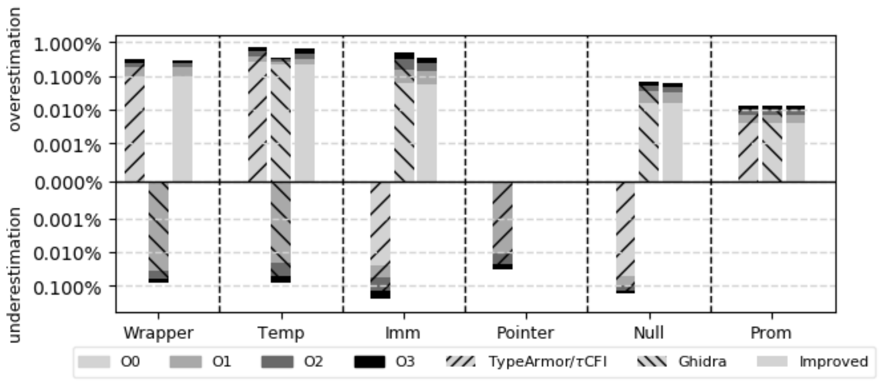


(c) C applications compiled by GCC

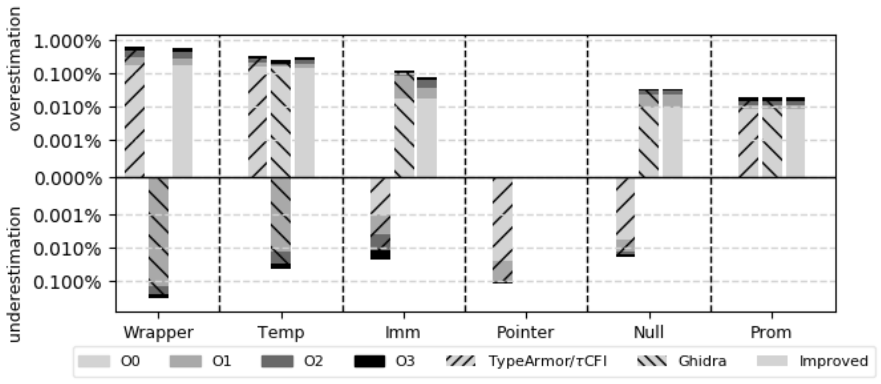


(d) C++ applications compiled by GCC

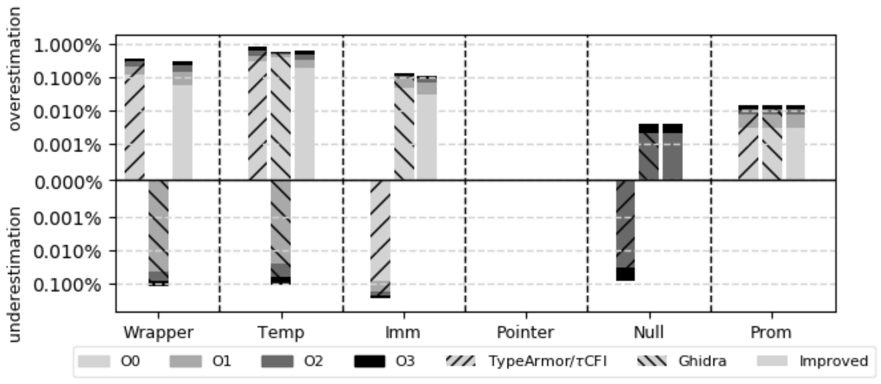
Figure 3.1: Likelihood of complications at callees



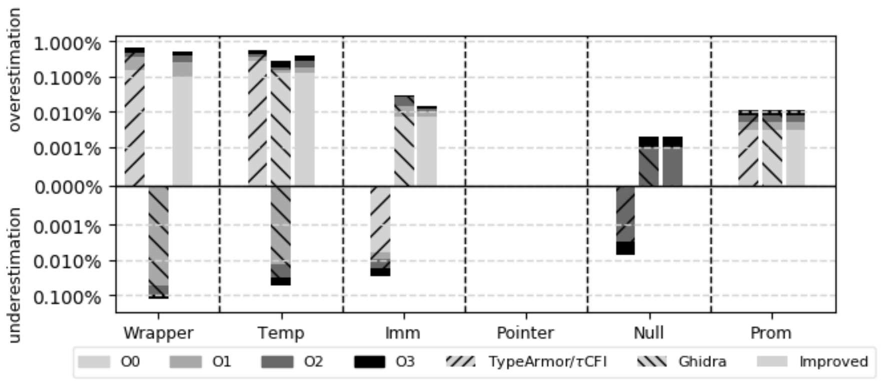
(a) C applications compiled by Clang



(b) C++ applications compiled by Clang



(c) C applications compiled by GCC



(d) C++ applications compiled by GCC

Figure 3.2: Likelihood of complications at callers

Table 3.6: Number of functions and indirect calls

		Opt	#func	#normal func	#variadic	#icalls
clang	C	O0	543	486	21	64
		O1	540	495	19	60
		O2	394	346	19	85
		O3	380	352	19	93
	C++	O0	3,379	3,229	15	121
		O1	3,290	3,085	13	74
		O2	702	652	13	439
		O3	710	640	13	452
gcc	C	O0	546	483	24	70
		O1	446	420	19	69
		O2	418	386	21	67
		O3	406	370	22	83
	C++	O0	4,505	4,113	22	152
		O1	686	608	13	336
		O2	698	613	13	312
		O3	656	606	13	299

natures at callees, where the fact that many functions do not read (some of) their arguments leads to underestimation of the number of arguments. It also potentially leads to underestimation of the width of an argument register whose evident reading instruction is missing while existence is implied (due to subsequent argument registers whose reading instructions being present). This complication presents more heavily in C++ programs due to the simplicity of many (callee) functions whose implementation does not require accessing the **this* argument. Another finding is that C++ applications compiled by `gcc` tend to have dead code eliminated, which makes them seemingly less vulnerable to this complication. Note that unoptimized binaries do not have this issue at all because compilers always insert argument reading instructions even if the callee function does not need them.

***Nor2Var*:** This also presents heavily in our test suite, leading to underestimation on the number of arguments, especially in C programs, except that compiler optimization actually helps mitigating it. As explained in Section 3.1.2, unoptimized binaries always move all arguments onto the stack, making it more likely to present more than 5 integer arguments at the callee which always leads to misidentification of variadic functions. Optimization helps “skipping” some of the arguments and reducing the likelihood of misidentification. Ghidra is immune to this complication since it simply considers all functions non-variadic.

lea*, *Push*, and *Prom*:** These three complications result in overestimation on the argument width, and together present a large thread to function signature identification of optimized binaries. Checking into the details, we find that C programs make heavier use of `lea` to perform simple computations and more often push arguments onto the stack (especially with `gcc`). Looking into the case of ***Prom, we find that `clang -O0` does not promote the argument width (it uses register `al` or `ax` to store the argument) while `gcc` does (it uses `eax`) even when optimization is turned off.

***rdx*:** This presents more on C++ programs and leads to overestimation on the number of arguments. Upon checking the details, we realize that the exception handling in C++ will call function `rethrow_exception`, which invokes function `_Unwind_RaiseException` that returns the unwind reason code in `%rdx` and the exception object in `%rax`.

***Var2Nor*:** As expected, Ghidra is vulnerable to this, although not that much due to compiler optimization but the simple treatment it employs (all functions are non-variadic). This complication presents to TypeArmor and τCFI, and is usually due to empty implementation of functions with more than five compulsory arguments. We find that C programs compiled by `gcc` suffer overestimation on the number of arguments on top of function type misidentification.

***VarOver*:** This only presents itself on binaries compiled with `gcc -O2` and `-O3`, where the instructions that move the variadic arguments onto the stack are not back to back. On the other hand, all variadic functions are identified as non-variadic in Ghidra, so the number of arguments is overestimated.

Temp* and *Wrapper*:** These are clear examples in which compiler optimization helps TypeArmor and τCFI determining the number of function arguments. In the case of ***Temp, optimization eliminates redundant instructions as function arguments. ***Wrapper*** causes fewer complications in optimized binaries due to heavier applications of function inlining. Note that C++ applications are more vulnerable to

Table 3.7: Likelihood that indirect calls in C programs use immediate values as arguments

Compiler	Opt	Non-inline	Inline	Loop-unroll	Func-copy
clang	O0	24	0	0	3
	O1	23	0	0	6
	O2	13	52	0	3
	O3	13	49	45	4
gcc	O0	81 (25)	0	0	3
	O1	75 (23)	59	0	5
	O2	26	28	0	3
	O3	22	33	3	3

Numbers in brackets correspond to functions that pass the value 0 to an argument register.
"Func copy" refers to multiple copies of the same function called from different modules.
Results shown are likelihood results multiplied by 1,000, rounded to the nearest integer.

Wrapper due to the large number of virtual functions being called indirectly. Ghidra generally performs worse here (considering the combined errors in both over- and underestimation) mainly due to its limited scope of backward analysis for indirect calls in wrapper functions. That said, Ghidra has superior mechanisms in dead code elimination and only the basic block which contains an indirect call is analyzed, which results in some argument registers that are used for temporary storage being correctly identified; see the complication of ***Temp*** (overestimation).

Imm* and *Null*:** C applications compiled with `clang` and `gcc` are both likely to pass immediate values to argument registers, which results in underestimation of the argument width by TypeArmor and τCFI. Interestingly, the likelihood increases upon increase of optimization levels. Digging into the details, we realize that this is actually just an artifact because higher optimization level results in heavier application of function inlining (`-O1` and `-O2` for `clang`, `-O1`, `-O2`, and `-O3` for `gcc`) and loop unrolling (`-O3` for both compilers), which leads to a larger number of callers of the same function; see Table 3.7 which shows the likelihood that indirect calls use immediate values as arguments for different reasons. Another interesting observation is that `gcc -O0` and `-O1` are more likely to move zero (Null***) to an argument register than using `xor`.

Ghidra, on the other hand, is not vulnerable to this underestimation but rather suffers on overestimation because it always uses the entire 64-bit memory range as the argument width.

***Pointer*:** This only affects applications compiled with `clang` especially on C++ programs as they are more likely to pass pointers to indirect callees. C programs compiled with `clang -O0` do not have this problem because it uses a 64-bit register to store the pointer by adding a prefix to denote the use of a 64-bit displacement or immediate source operand. C++ programs, on the other hand, set a 32-bit register to the pointer address and then move it to the argument register for some indirect calls. We also find that C++ applications compiled with `clang -O1` have a higher likelihood on this complication. This is because for some indirect calls that accept pointers as arguments, `clang` prepares them by moving 64-bit immediate values onto the stack first, and then after another indirect call instruction, the argument register is set by reading the 64-bit value from this stack address. As the number of indirect calls in binaries compiled with `-O2` and `-O3` is much larger, the likelihood for them becomes smaller.

Ghidra, again, is not vulnerable to this because it always uses the entire 64-bit memory range as the argument width.

Applications compiled by `gcc` do not use pointers that point to .text, .rodata, or .bss as arguments because `gcc-7` and above compile applications into position-independent code.

***Prom*:** This seems to be less sensitive to compiler optimization (compiler will always promote to the native type — 32 bits) and only affects a small number of indirect calls.

3.1.4 Our Compiler-Optimization-Friendly Policies

In an effort to properly handle the complications arisen due to compiler optimizations to more accurately recover function signatures, we propose a set of improved policies. In this section, we first discuss the details of these policies and then present our evaluation results of applying them to analyze our test suite of 1,344 real-world applications. Note that most of the policies proposed here are generally accurate for both optimized and unoptimized binaries, while others are more specifically targeting optimized binaries. Existing work [69, 70] and our experience (e.g., if values of all six argument registers are moved onto the stack, then it must be an unoptimized binary) show that detecting the compiler and the optimization level used in well-behaved binaries can be done accurately, and we take it as a prerequisite of enforcing our policies specifically targeting optimized binaries.

Identifying Variadic Functions (Targeting *Nor2Var* and *VarOver*)

The main problem in existing approaches is the identification of variadic arguments using "back-to-back value assigning instructions" (i.e., $b2b$) [85], which is not a sufficient condition as we analyzed (see Section 3.1.2) and showed in experiments (see Section 3.1.3). We discover another more direct and sufficient condition for variadic argument identification when optimization is enabled, in which the stack addresses storing variadic arguments are consecutive, prepared using 64-bit registers, and read using pointers. More specifically,

Definition 12. *Let* $@_i$ *denote the stack address to which argument register* i *is moved given* $r\dot{w}2s_i()$. *Callee function* f *is a variadic function iff* $\forall i \in \{5,4,3,2,1\}$,

- $|@_{i+1} - @_i| = 8$*; and*
- $s^{EE}_{i+1} = r\dot{w}2s_{i+1}(64)$ *and* $s^{EE}_i = r\dot{w}2s_i(64)$*; and*
- $@_{i+1}$ *and* $@_i$ *are read via pointers.*

with $|P^{OB}_{EE-f}|$ *being the maximal* i *violating the above. Otherwise,* f *is a normal*

Table 3.8: Analysis of the non-variadic function in Binutils

Line Number	Operation	TypeArmor	Our improved policy
4	Move %r9 to stack 0x40(%rsp)	%r9 is a variadic argument	May be a variadic argument
5	Move %r8 to stack 0x10(%rsp)	%r8 is a variadic argument	Non-consecutive stack addresses; not a variadic argument
11	0x40(%rsp) is read not overwritten		Not a variadic argument
Conclusion		Variadic function with 4 arguments	Normal function with 6 arguments

function and $|P^{OB}_{EE-f}|$ *is:*

$$\begin{cases} 6 & \textit{if } r\dot{w}2s_6() \textit{ and } @_6 \textit{ is not read via a pointer} \\ \max(\underset{i}{\operatorname{argmax}}(r\dot{w}2s_i()), \underset{i}{\operatorname{argmax}}(r\dot{w}_i())) & \textit{if } s^{EE}_6 \neq r\dot{w}2s_6() \end{cases}$$

We use the example in Listing 3.1a to show how our policy works. During analysis, we find that $P^{OB}_{EE-0x471a60} =< r\dot{w}_1(64), r\dot{w}_2(64), r\dot{w}_3(64), r\dot{w}_4(64), r\dot{w}2s_5(64), r\dot{w}2s_6(64) >$ and $@_6$ is not read via a pinter; therefore, we conclude that $|P^{OB}_{EE-0x471a60}| = 6$. Note that although $|P^{GT}_{EE-0x471a60}| = 7$, $|P^{OB}_{EE-0x471a60}| = 6$ is an accurate and best approximation based on the limited information present in the binary. The details about the analysis result by TypeArmor and our new policy can be found in Table 3.8.

The policy described above does not work well when optimization is disabled, in which all arguments are copied onto the stack at consecutive addresses. The policy to deal with unoptimized binaries is described in Definition 13.

Definition 13. *Callee function* f *is a variadic function iff* $\forall i \in \{5, 4, 3, 2, 1\}$,

- $|@_{i+1} - @_i| = 8$*; and*
- $s^{EE}_{i+1} = r\dot{w}2s_{i+1}(64)$ *and* $s^{EE}_i = r\dot{w}2s_i(64)$.

with $|P^{OB}_{EE-f}|$ *being the maximal* i *violating the above. Otherwise,* f *is a normal function with 6 arguments.*

Argument (Width) Promotion and Demotion (Targeting *Push*, *lea*, *Imm*, *Pointer*, and *Null*)

Our improved policy solves the argument promotion and demotion complications by analyzing the context of the instructions. More specifically,

- ***Push***: Let $p = 32$ in $r\dot{w}_i(p)$ if the corresponding argument reading instruction is `push`.
- ***lea***: Let p in $r\dot{w}_i(p)$ be the minimum of the width of the source and destination registers (instead of that of the source only as in TypeArmor and τCFI).
- ***Imm***: Let $p = 64$ in $r\dot{w}_i(p)$ if register i holds a constant.
- ***Pointer***: Let $p = 64$ in $r\dot{w}_i(p)$ if register i holds a pointer value pointing to .rodata, .bss, or .text section.
- ***Null***: Let $p = 64$ in $r\dot{w}_i(p)$ if register i is involved in an `xor` instruction.

Note that this improved policy guarantees that all legal callers be matched with legal callees since there is no underestimation at callers or overestimation at callees, but could lead to some imprecise (but conservative) results. For example, demoting the argument width to 32 bit for a register read using `push` may result in underestimation; see the case of ***Push*** in Figure 3.1. We believe that this is a good tradeoff where an absolutely precise solution does not exist, especially since the intended control flow is never broken with our improved CFI policy.

Register Overloading (Targeting *rdx*)

Since the overloading of `rdx` is for storing function return values, we simply consider any first reading of `%rdx` after a call to a library function (let's denote the callee f) as $w_3()$. It may first sound counter-intuitive, but this must be reading the return value of f since the compiler has to make a conservative assumption that f has reset `%rdx`. This improved policy solves the complication ***rdx*** at callees with 100% accuracy.

Registers Storing Temporary Values (Targeting *Temp*)

Recall that the analysis of callers considers all instructions involving an argument-passing register instead of focusing on only the first instruction (Section 3.1.1). Although that is technically correct, it also introduces complications since registers storing temporary values could be miscounted as passing arguments to a callee (***Temp***). Our improved policy takes into consideration the reading of registers (rather than focusing only on writing in the original policy) as well as the sequence of the instructions. More specifically, we let $s_i^{ER} = \hat{w}_i$ if register i is moved to another argument register after the write operation when the value of register i is not zero (a special case where the compiler will directly move register i to another argument register since the compiler does not prefer passing zeros to a register directly).

For example, as shown in Listing 3.7a, `%rcx` is moved to `%rdx` at Line 6 after the write operation at Line 4. With this, we conclude that `%rcx` is not used to pass arguments and $|P^{OB}_{ER-0x461015}| = 3$.

In order to be conservative, we only apply this policy to basic blocks where indirect calls are located. Note that this policy can also help correctly recover the number of arguments for indirect calls in wrapper functions.

Additional Binary Analysis to Extract our Policies

We have presented *what* our improved CFI policies are so far in this section. Here we briefly discuss *how* it is done with the additional binary analysis we perform.

Our improved policy for ***Nor2Var*** requires that we trace the data flow of a stack memory to check whether it is read without being overwritten. This is done by following the CFG of a function and check whether the stack memory is used as the source operand without being used as a destination operand.

Our improved policy for ***Imm*** requires that we identify whether one register holds a constant. Specifically, during the backward analysis, if we encounter a 32-

bit argument register being written to, we will record its source recursively and check whether it is an immediate value. Our experiences show that this recursive tracing typically reports a success within the same basic block and does not result in excessive overhead.

Evaluation of our Improved Policies

We apply our new policies on the same test suite consisting of 1,344 C and C++ applications and use the same metric as described in Section 3.1.3 to evaluate it; see the bars named "Improved" in Figure 3.1 and Figure 3.2. The comparison shows that our new policies result in significant improvement over most of the complication cases. In particular, we completely mitigate the complication cases of ***VarOver***, ***rdx***, ***lea***, and ***Pointer***, and significantly reduce the chances of running into ***Nor2Var***.

For cases of ***Imm***, ***Null***, and ***Push***, our policy guarantees that valid calls are never inadvertently blocked, but it could also potentially make the recovered function signatures more conservative. For example, we promote the argument width at indirect callers for cases ***Imm*** and ***Null***, which may result in overestimation on argument widths as shown in Figure 3.2 with likelihood less than 10.1% and 1.7%, respectively. Similarly, our policy to deal with ***Push*** may cause argument width underestimation at the callees, and the likelihood is about 0.2%. This raises an interesting question whether it is possible for CFI policies recovered from binary executables to be more accurate and approach the accuracy of source-based solutions; we discuss this in Section 3.1.5.

For ***Nor2Var***, the likelihood of misidentifying normal functions to variadic for unoptimized binaries is reduced from 3.3% to 1.2%, with that for optimized binaries dropped to 0.1%.

Since we only apply the policy for ***Temp*** to basic blocks where indirect calls are located, there can be overestimations if the argument registers storing temporary values are in other predecessors. The same policy also helps identify the number of arguments for indirect calls in wrapper functions as shown in the case of ***Wrapper*** in Figure 3.2 — the likelihood of overestimation on the number of arguments is reduced from 11.5% to 5.4% for C applications compiled by `gcc -O0`.

Potential revisions to deal with other complications

To handle ***Var2Nor***, we could revise our policy on identifying variadic functions to find the argument register with the highest index i that is moved onto the stack. However, this will result in (potentially unnecessary) checking of registers at a smaller index, and lead to substantially higher overhead in the processing. Since we only observe one variadic function (`bfd_set_error` in `Binutils`) being misidentified as a normal function and causing overestimation on the number of arguments in our large test suite, we do not suggest enforcing this policy.

Similarly for ***Unmodified***, we could perform backward analysis from the indirect caller until another indirect call is encountered. We do not enforce this policy because there is only one application in our test suite that has this problem (with only two indirect calls), and this policy could result in a large number of overestimation on the number of arguments at indirect callers.

3.1.5 Discussions and Security Implications

In this section, we first discuss an interesting question whether policies recovered from binary executables could approach the accuracy of source-based solutions, and then further evaluate the security implications of having inaccurate CFI policies.

Comparison with Source-Level Solutions

Section 3.1.4 shows that even our improved policy inevitably results in some over- and underestimation, which raises an interesting question whether it is possible to further improve the policies so that their accuracy approaches that of source-level solutions. Here we present three scenarios where a compiler makes the task of accurately recovering function signatures undecidable, and therefore show that binary-

```
1 5a0d32: xor    %esi,%esi
2 5a0d34: xor    %edx,%edx
3 5a0d36: mov    %rbp,%rdi
4 #struct ref *(*get_refs_list)(struct transport *transport,
    int for_push, const struct argv_array *ref_prefixes);
5 5a0d39: callq  *0x10(%rax)
```

Listing 3.10: Immediate zero and NULL as arguments

level techniques can never achieve the accuracy of source-based solutions.

Immediate value zero vs. NULL pointer

A simple example demonstrating the limitation of binary analysis in this context is the differentiation between an immediate value zero and the NULL pointer. Line 4 of Listing 3.10 shows a callee function with the second and third arguments being integer and pointer type, respectively, while Line 1 – 2 show the caller preparation with identical instructions for these two arguments. It clearly demonstrates that binary analysis is unable to distinguish the two cases and would have to make approximations in recovering the caller signature.

Arguments unused

Another scenario arises in the case of unused arguments at the callee (corresponding to complication case ***Unread***), where binary analysis cannot differentiate

- Listing 3.11a: a callee function with an argument passed in but the argument is not used; and
- Listing 3.11b: a callee function without arguments.

Binary analysis would not be able to differentiate the two cases as observations on their argument-passing registers are identical.

Registers overloading

Registers are used for passing arguments as well as any other general purposes (corresponding to complication case ***Temp***), and binary analysis usually cannot dis-

```
bfd_plugin_core_file_failing_signal (bfd *abfd )
482000: push   %rax
482001: mov    $0x4dc9e1,%edi
482006: mov    $0x1ac,%esi
48200b: callq  405230 <bfd_assert>
```

a: Argument passed in but not used

```
void bfd_section_already_linked_table_free ()
48aa60: mov    $0x7172f8,%edi
48aa65: jmpq   406860 <bfd_hash_table_free>
```

b: No argument

Listing 3.11: Function argument unused

```
51e199:mov  %eax,%esi
51e19b:test %r15,%r15
51e19e:je   51e1ad
51e1a0:lea  0xe0(%rsp),%rdi
#(fptr_T)(func_one(&cc, c));

51e1a8: callq *%r15
```

a: %esi used to pass argument

```
43ae62: mov   %ebp,%esi
43ae64: test  %rax,%rax
43ae67: je    43ae6f
43ae69: mov   %ebp,%edi
#get_elf_backend_data(abfd)->
    obj_attrs_order(i);
43ae6b: callq *%rax
```

b: %esi used to store temporary

Listing 3.12: Example of argument register usage

tinguish the two cases. Listing 3.12 shows two indirect callers with

- Listing 3.12a: a caller that uses `%esi` to pass the second argument to callee.
- Listing 3.12b: a caller that uses `%esi` to store a temporary value.

Again, binary analysis would not be able to tell apart these two cases and an approximation has to be made in extracting function signatures.

We stress that this is not an exhaustive list of cases where binary analysis may fail, but the three scenarios identified are specific to funciton signature recovery where compiler optimization makes binary analysis *undecidable*.

Security Implication with Imprecise Function Signature Recovered

The undecidability in binary analysis results in inevitable errors in function signature recovery from (optimized) binary executables. An immediate question, therefore, is on the extent to which such errors impact security applications. In this subsection, we evaluate this security implication from two perspectives.

Imprecision on the set of callees allowed Our first evaluation focuses on the number of callees allowed in a CFI enforcement, and here we consider six solutions:

- AT [97]: A binary-level solution that allows indirect callers to target any "Address-Taken" functions;
- TypeArmor [85]: A binary-level solution with function signatures capturing the number of arguments;
- τCFI [55]: A binary-level solution with function signatures capturing the number of arguments and width of arguments;
- Our improved policy: A binary-level solution with function signatures capturing the number of arguments and width of arguments, targeting optimized binaries; and
- IFCC [84]: A (relatively old) source-level solution with function signatures capturing the number of arguments; in LLVM-3.4.
- LLVM-CFI[8]: A (latest) source-level solution with more precise function signatures (the number of arguments and their primitive types, function return type) captured; in LLVM-10.0.

Table 3.9 shows the median of the number of callees allowed for each indirect caller for the 1,344 applications in our test suite under different policies. We can see that compared to AT, TypeArmor, τCFI, and our improved policies reduce the number of legal control-transfer targets by about 20%, 49%, and 54%, respectively, while none of the binary-level solutions could achieve precision of source-level techniques. In particular, LLVM-CFI achieves much better accuracy because

[8]`https://clang.llvm.org/docs/ControlFlowIntegrity.html`

Table 3.9: Number of callees allowed by different policies

		Opt	AT	TypeArmor	τCFI	Improved	IFCC	LLVM-CFI
clang	C	O0	543	412	290	246	114	7
		O1	540	446	242	213	124	8
		O2	394	318	147	147	93	7
		O3	380	300	130	120	99	8
	C++	O0	3,379	2,734	2,343	2,186	1052	37
		O1	3,290	2,631	1,879	1,805	998	35
		O2	702	552	304	270	251	44
		O3	710	543	296	284	247	44
gcc	C	O0	546	499	336	257		
		O1	446	373	272	239		
		O2	418	318	147	147		
		O3	406	332	231	200		
	C++	O0	4,505	3,920	3,278	3,219		
		O1	686	498	314	301		
		O2	698	477	294	281		
		O3	656	527	315	299		
Geomean			767	612	395	353	232	19

it uses finer-grained types of arguments — char* and const char*, struct A* and struct B* are considered different types — which cannot be differentiated at binary level.

Effectiveness in allowing/disallowing COOP gadgets With Table 3.9 showing the number of mistakes each solution makes, we next evaluate the extent to which these mistakes result in initial COOP gadgets an attacker could use to construct code-reuse attacks. This time, we only focus on τCFI and our improved policy as they run relatively close in the previous evaluation. We use the same heuristics proposed in the corresponding papers to find potential Main-Loop Gadgets (ML-G) [71] and RECursive Gadgets (REC-G) [26] for all C++ applications in our test suite. Table 3.10 shows the total number of such gadgets as well as the number of such gadgets whose function signatures are correctly identified by τCFI and our improved policy. Bigger numbers indicate better effectiveness of CFI in disallowing the corresponding code-reuse attacks.

As we can see, τCFI correctly identifies 68% and 64% ML- and REC- gadgets, respectively, while our improved policy achieves 78% and 74% effectiveness, respectively. We believe that this evaluation provides a good indicator on the security

Table 3.10: Potential ML-G and REC-G gadgets

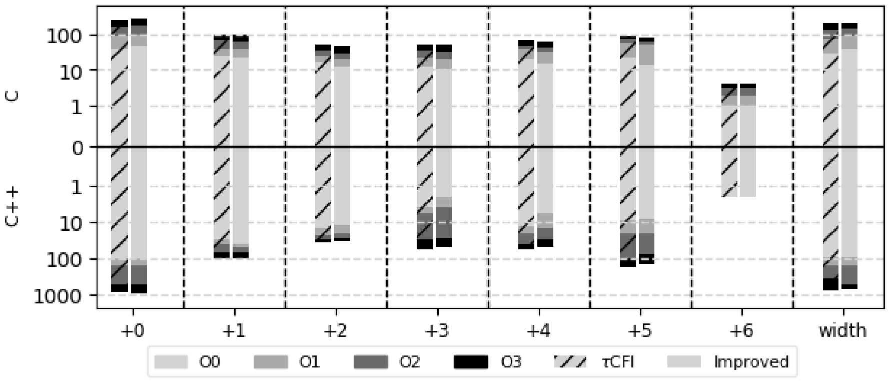
	Opt	ML-G			REC-G		
		icall	τCFI	Improved	icall	τCFI	Improved
clang	O0	93	53	64	73	41	45
	O1	58	50	50	56	44	44
	O2	70	46	52	60	41	44
	O3	70	42	53	49	35	39
gcc	O0	96	50	68	71	32	46
	O1	98	71	80	74	50	56
	O2	113	100	103	33	21	30
	O3	106	79	84	22	15	17
Geomean		83	56	65	58	37	43

impact of our improved CFI policies.

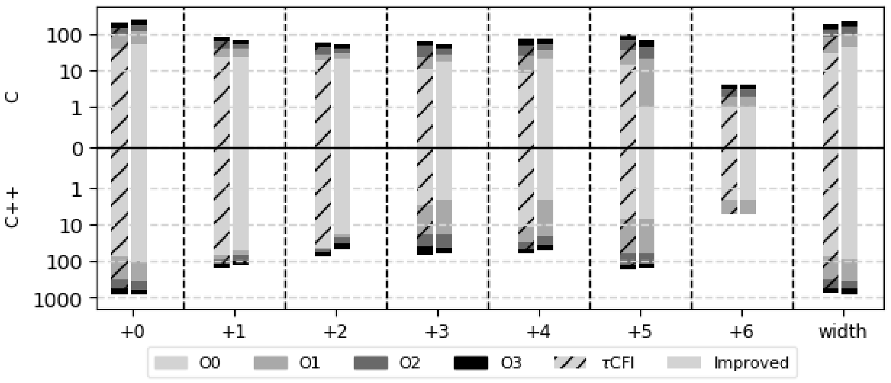
Severity of each mistake For each mistake in recovering function signature of the caller, we check how far the mistake is from the ground truth, which also has a direct implication on the amount of flexibility an attacker has when using the corresponding caller to construct an code-reuse attack. Figure 3.3 shows the result of this evaluation, again, on our test suite of 1,344 applications, with x-axis labels being:

- $+t$: the average number of indirect callers whose number of arguments is overestimated by t; and
- width: the average number of indirect callers whose function signature (number and width of arguments) is correctly recovered.

Besides showing the consistently better results from our improved policy compared to those from τCFI, we also notice that our improved policy performs most significantly better on "+5", which means our improved policies manage to correct a larger number of more severe mistakes made by τCFI.



(a) Applications compiled by Clang



(b) Applications compiled by GCC

Figure 3.3: Amount of flexibility of code-reuse attacks in each mistake in function signature recovery of indirect callers

3.2 Enhanced Deep Learning Approach to Recover Function Signature

In Section 3.1, we introduced that compiler optimizations have a significant impact on function signature recovery, and the analysis engines need to be continuously updated so that they can be used to analyze binaries compiled by these new versions of compilers. EKLAVYA [20] attempted to recover the function signature by using a Recurrent Neural Network (RNN) to process the raw binary code without relying on the calling convention. It introduced a new way to learn function signature and can obtain better accuracy when compared to these calling-convention based approaches. However, some intricacies make the accuracy of EKLAVYA relatively small (e.g., the reported accuracy in identifying the number of arguments for optimized binaries at the callee site is less than 80% on x86-64). We find the main limitation of EKLAVYA is that it didn't incorporate domain knowledge to improve the quality of the dataset. For example, if function a has three arguments, but only the first two are used, EKLAVYA still labels it has three arguments. This kind of labeling could mislead the training process to study function signature incorrectly.

In this section, we first study the possible intricacies that EKLAVYA didn't solve and then propose solutions to mitigate them. The result shows that our approach can effectively improve the accuracy in identifying the number of arguments from the callee.

3.2.1 Workflow of EKLAVYA

The workflow of EKLAVYA is shown in Figure 3.4. We can find the first essential step is to uncover each instruction's semantic information through learning by using *word embedding*. Specifically, it takes as input a stream of instructions and outputs a vector representation of each instruction in a 256-dimensional space by using `word2vec` [53]. The generated sequence of vectors (each representing an instruction), together with labels denoting the number of arguments and types (the ground truth) are used as the input to a three layers recurrent neural network with gate recurrent units (GRUs). Specifically, for argument type recovery, EKLAVYA learns one RNN for the first argument, one RNN for the second argument, and so on.

There are four tasks in EKLAVYA:

- **Task1**: Counting arguments for each function using instructions from callers;
- **Task2**: Counting arguments for each function using instructions from callees;
- **Task3**: Recovering the type of arguments based on instructions from callers;
- **Task4**: Recovering the type of arguments based on instructions from callees;

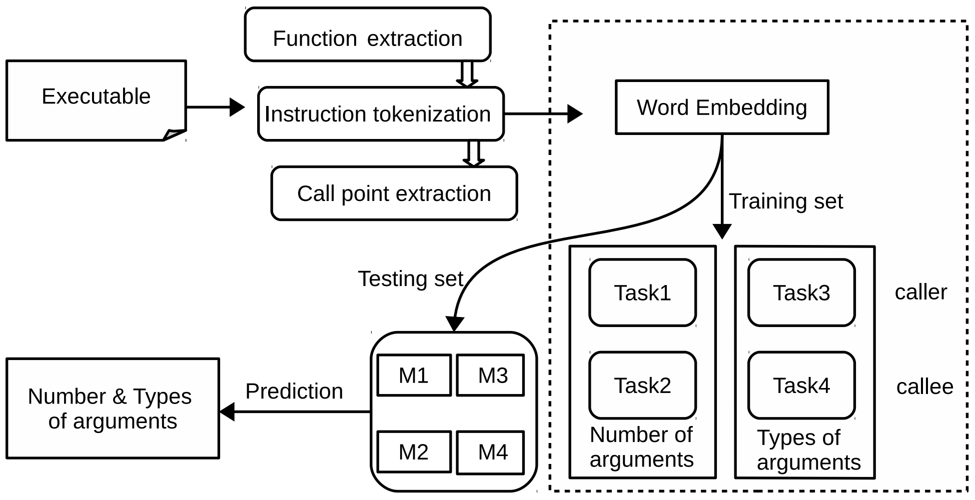


Figure 3.4: Workflow of EKLAVYA

The classes of argument types are defined as $\tau ::= int|char|float|void * |enum|union|struct$. That is to say, it treats different kinds of integers (e.g., 32-bit integers and 64-bit integers) to be class int.

In task two and four, instructions in the callee itself will be used. All the instruction preceding a direct `call` instruction is used in task one and three. For large functions, it limits the number of instructions to 500.

We can find EKLAVYA directly uses the instruction in the function body as the input to the neural network without considering whether they actually access the argument registers (memory) and whether they are distinguishable. We will introduce possible intricacies that we identified and aren't solved by EKLAVYA in detail in the following sections.

3.2.2 Intricacies and Solutions

- **I1: Missing argument-reading instructions.** It includes the access of an argument is in other functions, and some function arguments are not used.

  - **Arguments are accessed in other functions (denoted as *Prop*).** There are cases that the access of an argument would be in other functions when optimization is enabled. EKLAVYA only uses instructions in the function body to

```
LUA_API int lua_toboolean (lua_State *L, int idx) {
  const TValue *o = index2adr(L, idx);
  return !l_isfalse(o);
}
0000000000402650 <lua_toboolean>:
402650: push   %rax
       : d6 f6
       : 48 d6 4f 10
402651: callq  4021f0 <index2adr>
402656: mov    %rax,%rcx
402659: mov    0x8(%rax),%eax
40265c: test   %eax,%eax
40265e: je     402674 <lua_toboolean+0x24>
402660: cmp    $0x1,%eax
402663: jne    40266f <lua_toboolean+0x1f>
402665: xor    %eax,%eax
402667: cmpl   $0x0,(%rcx)
40266a: setne  %al
40266d: pop    %rcx
40266e: retq
......
```

a: Example of arguments are accessed in other function body

```
Instructions access the arguments
4021f0: 85 f6          test   %esi,%esi
402206: 48 3b 4f 10    cmp    %rcx,(%rdi+0x10)
```

b: Argument reading instructions

Listing 3.13: Arguments are accessed in other function body

infer the function signature, which may not get the correct result. As shown in Listing 3.13a (without instructions in the gray background), all arguments of function `lua_toboolean` are accessed by function `index2adr`. In this case, EKLAVYA still says it has two arguments but without any evidence to show that it access any argument in the instructions that inputted to the RNN. It would confuse the training process in identifying the number of arguments.

One possible solution is to include all instructions that follow the inter-procedural control-flow graph, but it will also include many irrelevant instructions that may affect the training result. Instead, we make use of TypeArmor to find all instructions that read the argument register by following the control-flow graph and then insert our special instructions before the call instruction.

For the example in Listing 3.13a, we find two instructions in function `index2adr` access the argument register, which are shown in Listing 3.13b. Since we only find argument register reading instructions, they cannot be directly used to prove that this function has two arguments. Therefore, we do not directly insert them into function `lua_toboolean`, but using our special instructions to tell the training process that they are inserted by us. Specifically, we replace these instructions' opcodes with unused opcodes defined in Intel Manual [41]. Different kinds of opcodes will be replaced by different values.

  ∗ One-byte opcode. `0xd6` is used to replace it.
  ∗ Two-byte opcode and the operand is integer. `0x0f 0x25` is used to replace it.
  ∗ Two-byte opcode and the operand is floating-point. `0x0f 0x27` is used to replace it.
  ∗ Three-byte opcode and the operand is integer. `0x0f 0x38 0x51` is used.
  ∗ Three-byte opcode and the operand is floating-point. `0x0f 0x38 0x53` is used.

The inserted special instructions are shown in Listing 3.13a with gray background. Now we will use the new function with inserted instructions to perform training. Similarly, we also do this kind of analysis for the testing set and insert our special instructions.

– **Arguments are not used (denoted as *Unread*).** There are some cases that we have to want an unused argument, usually due to the function has to conform to a fixed prototype. For example, when it is virtual or it is going to be called from a template. However, the label provided by EKLAVYA in the training set always contains these unused arguments, which may confuse the training process. As shown in Listing 3.2, the first and third arguments of `jpeg_free_large` are not used, but the label used in the training set is it

```
00000000000342a8 <ar_emul_append>:
        : 48 of 25 c7            op    %rdi
        : 48 0f 25 c6            op    %rsi
        : 48 0f 25 c2            op    %rdx
        : 0f 25 c1               op    %ecx
        : 41 0f 25 c0            op    %r8d
342a8: 48 8b 05 19 8e 2f 00     mov   0x2f8e19(%rip),%rax
342af: 48 85 c0                 test  %rax,%rax
342b2: 74 0b                    je    342bf
342b4: 48 83 ec 08              sub   $0x8,%rsp
342b8: ff d0                    call  352b0
```

Listing 3.14: Inserted instructions at caller sites

has three arguments. In this case, we will correct the label so that now the training model tells us how many arguments are used by a function rather than how many arguments it has. Therefore, in the training set, we label function `jpeg_free_large` uses two arguments.

For the unused arguments that do not have the attribute `__attribute__((unused))`, we make use of an LLVM pass to correct the label at the callee and caller site. Otherwise, we perform static binary analysis based on TypeArmor to check how many argument registers are actually used and correct the corresponding labels, the call sites that call those functions do not need to be corrected as they always set the argument.

I2: Missing argument-setting instructions (denoted as *Wrapper*). As described in TypeArmor [85], if a caller is in a wrapper function, the compiler may not need to reset all argument registers but simply 'pass them through' directly from its caller when optimization is enabled.

As shown in Listing 3.14, EKLAVYA only uses these five instructions (instructions in white background) to infer the number of arguments for the caller at address `0x342b8`. We can find none of them access the argument register, but the input to the RNN is it has five arguments. It may cause EKLAVYA to infer the number of arguments for callers incorrectly.

For this case, we may perform the backward analysis and find all possible argument setting instructions and then insert them into the wrapper functions. However, since all argument registers can also be used to store temporary values, this kind of backward analysis may include many "noisy" instructions. Therefore, instead of performing backward analysis, we insert our special instructions to represent which argument register is accessed by the callee of a call instruction.

For the example in Listing 3.14, we find the function at address `0x352b0` has five arguments, so we insert our special instructions for them at the entry of the wrapper function, and the inserted instruction is shown in gray background in Listing 3.14. We use the same instruction encoding mechanism used in Intel manual to encode our special instructions, the difference is we always use two-byte opcode `0x0f 0x25` for integer arguments and `0x0f 0x27` for floating-point arguments.

We perform static binary analysis based on TypeArmor without inter-procedural backward analysis to find all callers whose arguments are not set in the function body. Specifically, we identify how many argument registers are set for each caller and then compare it with the ground truth. We say a caller in the wrapper function does not set the argument if the static analysis result is smaller than the ground truth, then we will insert the special instruction for this caller.

I3: Irrelevant instructions (denoted as *Irrelevance*). EKLAVYA makes use of the entire instructions of a function as the input to the training process. However, there are many instructions that are unlikely to be related to the identification of the number (types) of arguments. This kind of "noise" may affect the performance of machine learning as shown in [76]. We want to make the deep learning approach perform better by eliminating "noise" in the training and testing set, allowing it to focus on the part of the binary that matters.

Therefore, we try to eliminate those irrelevant instructions. Specifically, we say one instruction is relevant if it accesses any argument register or stack address. Meanwhile, any branch instruction is considered to be relevant too. For exam-

```
46f019:  mov     0x8(%r12),%rax
46f01e:  mov     0x348(%rax),%rax
46f025:  mov     $0x1,%esi
46f02a:  mov     %r12,%rdi
46f02d:  callq   407f30
```

a: 32-bits immediate value

```
401c13:  mov     0x30(%r15),%rcx
401c17:  mov     $0x1,%esi
401c1c:  mov     $0x1000,%edx
401c21:  mov     %rax,%rdi
401c24:  callq   4020c0
```

b: 64-bits immediate value

Listing 3.15: Different integer types use the same instruction

ple, the irrelevant instructions for function `lua_toboolean` are shown in Listing 3.13a with dark-gray background.

I4: Undistinguishable cases

– **Argument registers are used for other purposes (denoted as *Temp*).** As described in Section 3.1.2, all argument registers could also be used as scratch registers to store temporary values. It may make the identification of the function signature at the caller site more difficult since the training process may not distinguish whether one register is used to pass argument or store temporary value.

– **Undistinguishable argument type (denoted as *Undis-type*).** EKLAVYA considers all kinds of integers (e.g., bool, short, int, and long) as type int; we also want to learn the exact type (width) for each argument so that this kind of information can be used to help CFI enforcement.

  However, there are cases that different kinds of integer types will use the same kind of instructions, especially for type int, long, and pointer.

  As shown in Listing 3.15, the types of the second arguments for these two callers are `int` and `long` respectively. However, we can find the same instruction `mov $0x1,%esi` is used to pass the second argument. It may confuse the training model to incorrectly infer a 64-bit argument to be 32-bit.

Since we cannot distinguish the actual argument type and the number of arguments, we propose to output the top five prediction results rather than only the top one. For the example in Listing 3.15b, the top five outputs for the type of the second argument are [$int32$, $int64$, $pointer$, $float$, $int8$] with probability [8.76e-01,1.24e-01,3.18e-05, 2.23e-05, 2.44e-07]. We can find it has a high probability that its type is $int64$ rather than $int32$. In the current implementation, in the top five outputs, if we find the probability of one output is not less than 10% and the predicted result is equal to the ground truth, we say the predicted result is correct.

With these solutions, our goal is to learn a model which is used to decide two properties for a target function a.

- **Number of arguments.** At the callee site a, it is the number of arguments used by it. At the caller site, it is the number of arguments passed to it.
- **Types of arguments.** For each argument of function a, it is defined as: $\tau ::= int8|int16|int32|int64|pointer|struct|float$

  Different from the $struct$ used in EKLAVYA, we only use $struct$ to represent the argument whose aligned size is larger than 16 bytes and must be passed onto the stack. We also don't have type $enum$ and $union$ used in EKLAVYA since the type of an enumeration (union) argument shall be the type of the container type. In other words, the actual type of an $enum$ ($union$) argument is always in τ.

3.2.3 Evaluation

In this section, we evaluate the accuracy in identifying the number and types of arguments. Our experiments are performed on a server containing 2, 32-core AMD ryzen threadripper 3GHz CPUs with 128GB of RAM and 4 GeForce RTX 2080 Ti GPUs with 12GB of memory. We use the GPU to perform the training. The neural network and data processing routines are written in Python, using the Tensorflow platform [2].

We base our ground truth on information collected by an LLVM [44] pass and on DWARF v4 debugging information [21] which is the default setting for `gcc` and `clang`. We use LLVM to collect source-level information, including the number and types of arguments for each function and callers. We also record the source line numbers of functions and callers. We then compile the test applications with DWARF information and link the source-level line numbers with binary-level addresses using the DWARF line number table. Different from EKLAVYA which obtains the ground truth by parsing the DWARF debug information, the ground truth we obtained is more fine-grained. For example, if one function has an argument whose type is a structure and the size of it is less than 16-bytes, it will be passed by two consecutive integer argument registers. In this case, the collected ground truth by LLVM is this function has two arguments rather than one.

Dataset

We use the same dataset used in EKLAVYA to generate the binary, including *binutils*, *coreutils*, *findutils*, *sg3utils*, *utillinux*, *inetutils*, *diffutils*, and *usbutils* but compiled with the latest compiler version. We only focus on function signature recovery in x86-64 Linux, and the binary is generated by using two commonly used compilers: gcc-10 and clang-10, with different optimization levels ranging from O0 to O3. Specifically, it contains 2,584 different binaries, 51,907 distinct functions, and 104,046 direct callers.

The statistics on the number of arguments and the type of argument are shown in Table 3.11 and Table 3.12 respectively. We can find most functions have less than three arguments. Therefore, we only infer the types for the first three arguments. For the first three arguments, most of them are pointers, 32-bit integers, and 64-bit integers.

We also reported the number of ***Prop***, ***Unread***, ***Wrapper***, and ***Temp*** in Table 3.13. We can find unoptimized binaries are more likely to use argument registers to store temporary values. In the dataset, the number of intricacy cases in binaries

Table 3.11: Number of arguments of functions in dataset

Opt	Number of Arguments (%)									
	0	1	2	3	4	5	6	7	8	9
O0	7.36	32.50	31.41	18.11	7.21	3.29	0.08	0.01	0.02	0.01
O1	9.58	30.26	30.46	17.27	6.72	3.27	1.37	0.58	0.37	0.12
O2	8.93	27.43	31.49	18.02	7.43	3.72	1.61	0.72	0.48	0.19
O3	7.76	21.50	32.97	20.12	9.38	4.53	2.20	0.86	0.45	0.24

compiled by “-O1” is more than binaries compiled by “-O2” and “-O3”.

We use 5-fold cross-validation to perform training. Specifically, we randomly split the identified intricacy cases in Table 3.13 into five folds (one used for testing, and the remaining is used for training). For other functions (callers), we randomly split each utility package into five folds. Note that the training set contains all binaries of one instruction set, compiled with multiple optimization levels from both compilers. The test results are reported on different categories of optimizations for different compilers.

Accuracy

Our goal is to evaluate the accuracy of prediction for the four tasks by using the approach we proposed in Section 3.2.2 to solve the intricacies we identified. The final results are shown in Table 3.14 and Table 3.15. Baseline means we use EKLAVYA to perform training and testing on the same dataset but without fixing the intricacy we identified.

We can find our approach effectively improve the accuracy in identifying the number of arguments at the callee site (Task 2) when optimization is enabled, especially for functions compiled by ”-O1”. This is because functions compiled with optimization ”-O1” has more cases we identified as shown in Table 3.13, our approach can improve more on it. According to the analysis result of the third model in the 5-fold cross-validation models, we find for case ***Unread***, our approach can correctly identify the number of arguments for more than 90% functions that are misidentified in EKLAVYA. For functions in case ***Prop*** whose number of arguments are misidentified in EKLAVYA, our approach can correctly identify the number of

Table 3.12: Types of arguments of functions in dataset

Opt	Type	Argument Index (%)								
		1	2	3	4	5	6	7	8	9
	int8	0.95	1.39	2.12	2.27	4.00	48.28	30.77	20.00	66.67
	int16	0.15	0.13	0.30	0.18	0.69	6.90	7.69	-	-
	int32	16.41	19.06	21.16	26.36	29.52	3.45	40.77	10.00	-
O0	int64	4.00	11.78	15.16	17.02	19.03	3.45	0	-	33.33
	pointer	78.32	67.46	60.96	53.91	45.38	17.24	30.77	70.00	-
	float	0.12	0.07	0.26	0.27	1.38	6.9	-	-	-
	struct	0.04	0.11	0.03	0	0	13.79	-	-	-
	int8	0.86	1.32	2.12	2.05	2.43	2.30	3.03	1.60	5.17
	int16	0.05	0.11	0.22	0.26	0.56	0.21	0.43	-	-
	int32	15.49	18.86	22.18	28.09	33.33	39.25	47.19	44.00	60.34
O1	int64	3.24	11.56	14.54	18.57	16.15	11.90	6.06	6.40	5.17
	pointer	80.20	68.00	60.68	50.85	47.06	45.51	43.29	48.00	29.31
	float	0.13	0.06	0.24	0.17	0.47	0.21	-	-	-
	struct	0.02	0.09	0.02	-	-	0.63	-	-	-
	8-bit	0.76	1.02	1.79	1.44	2.10	1.94	2.60	0	5.13
	16-bit	0.06	0.08	016	0.22	0.30	-	-	-	-
	32-bit	17.50	17.53	22.09	29.46	32.93	43.37	38.05	51.76	58.97
O2	64-bit	2.52	11.35	14.22	18.27	14.22	9.39	3.90	3.53	-
	ptr	79.10	69.91	61.45	50.61	49.85	44.66	45.45	44.71	35.90
	float	0.07	0.05	0.29	-	0.60	-	-	-	-
	struct	0	0.06	-	-	-	065	-	-	-
	8-bit	0.96	1.01	2.34	1.53	4.40	4.24	5.56	0	6.25
	16-bit	0.04	-	0.36	0.38	0.80	-	-	-	-
	32-bit	17.92	17.59	22.81	31.17	32.40	36.44	38.89	27.59	31.25
O3	64-bit	2.34	11.16	13.53	17.40	12.00	5.93	1.85	3.45	6.25
	ptr	78.70	70.18	60.78	49.33	49.60	53.39	53.70	68.97	56.25
	float	0.04	0.05	0.18	0.19	0.80	-	-	-	-
	struct	-	-	-	-	-	-	-	-	-

arguments for more than 50% of them.

The accuracy in identifying the number of arguments from the caller (Task 1) doesn't improve too much. We use the median model in the 5-fold cross-validation models to perform the analysis and find EKLAVYA can accurately identify the number of arguments for all callers which use argument registers to store temporary, but for other callers, it may misidentify that one register is used to store temporary value. This kind of inaccuracy can be mitigated by using the top five outputs. Mean-

Table 3.13: Number of intricacy cases

Opt	***#Prop***	***#Unread***	***#Wrapper***	***#Temp***
O0	-	-	-	6,803
O1	1,182	1,189	819	3,218
O2	349	568	82	1,630
O3	78	151	6	266
Total	1609	1908	907	11917

Table 3.14: Accuracy in identifying the number of arguments

Task	Opt	Our approach			Baseline		
		clang	gcc	total	clang	gcc	total
Task 1	O0	96.87	97.79	95.19	95.91	97.68	93.50
	O1	94.51	92.51		91.28	90.40	
	O2	96.07	90.80		93.32	88.57	
	O3	97.96	93.53		95.24	92.65	
Task 2	O0	99.22	98.74	95.80	98.95	98.46	92.20
	O1	92.67	93.62		84.80	87.50	
	O2	94.78	93.56		90.98	88.67	
	O3	95.96	95.45		94.60	91.73	

Table 3.15: Accuracy in identifying the type of argument

Task	CPL	Opt	Our Approach			Baseline			Our Approach (top1)		
			1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd
Task 3	clang	O0	97.34	94.44	93.78	96.94	92.97	92.51	96.65	92.54	92.16
		O1	97.26	93.83	93.29	96.78	92.30	92.67	96.71	92.09	92.00
		O2	98.88	96.88	96.63	98.02	95.62	96.59	98.28	95.26	95.63
		O3	98.44	97.93	98.30	97.97	97.16	96.56	98.05	96.37	98.14
	gcc	O0	97.40	93.09	90.73	97.33	91.54	91.05	96.72	90.56	88.16
		O1	97.03	92.20	91.17	96.39	90.61	90.83	96.12	89.26	89.23
		O2	96.27	91.74	91.57	95.76	91.10	91.42	95.25	88.95	89.39
		O3	97.58	93.79	93.68	98.03	94.01	94.77	96.95	92.28	92.40
Task 4	clang	O0	97.32	91.70	88.70	96.18	88.76	84.42	96.62	89.71	85.32
		O1	95.58	89.62	86.64	93.86	86.80	84.07	94.41	86.82	83.72
		O2	97.89	94.38	91.66	97.37	93.14	90.63	97.21	92.63	89.79
		O3	98.60	96.82	91.19	97.49	94.75	89.67	98.32	95.10	88.42
	gcc	O0	97.06	91.47	87.14	95.85	88.18	83.42	96.12	89.13	84.06
		O1	96.92	92.16	88.50	95.75	89.21	86.54	95.94	89.93	85.54
		O2	97.76	92.98	90.39	96.71	91.23	88.50	96.77	90.96	86.67
		O3	97.73	95.23	93.26	97.58	94.69	91.73	97.17	93.34	91.54

while, our approach by inserting special instructions for callers in wrapper functions makes the number of arguments for those callers all correctly identified (19 of them are misidentified in EKLAVYA). Since the number of cases for callers in wrapper functions is small, we believe if there are more cases, we can improve the accuracy more.

For the argument type recovery from the callee site (Task 4), we can find our approach improves the accuracy in identifying the type for the second and third arguments. This kind of improvement comes from the benefit of using the top five outputs. We also find the most inaccuracy comes from the misidentification between a pointer and a 64-bit integer. For the first argument, most of them are 32-bits integers and pointers as shown in Table 3.12, so the accuracy in identifying its type is much better.

The accuracy in identifying the type of an argument at the caller site by making use of our approach is comparable to the result of EKLAVYA. If we only use the top one output as the predicted label (the result is shown in the last three columns of Table 3.15), we can find the accuracy in identifying the argument type in a more fine-grained manner is much lower, especially for binaries compiled by gcc. This is because there are a large number of misidentifications between different kinds of integers and between 64-bit integers and pointers. For binaries compiled by clang, the main misidentification comes from the indistinguishability between integers and pointers.

3.3 Summary

In this chapter, we study how compiler optimization impacts function signature recovery implemented by TypeArmor and τCFI. Our study shows that compiler optimization has important impact on function signature recovery and potentially results in unmatched function signatures at callees and callers. In order to better deal with these optimizations, a set of improved policies is proposed, with results showing that most intricacies identified earlier being mitigated. Meanwhile, we propose an enhanced deep learning approach with domain knowledge included to recover function signature accurately.

The first part of this chapter is based on our previously published work [48] with no major changes.

Chapter 4

Control-Flow Carrying Code

4.1 Introduction

In the previous chapter, we introduce the approach to generate a more accurate CFG by making use of function signature matching, in this chapter, we will show how to implement the CFI policy securely.

An assumption made in most existing CFI approaches, including coarse-grained [97, 94] and fine-grained [57, 58, 84] ones, is that read-only data and code sections cannot be overwritten by attackers. For example, CFI proposed by Abadi et al. [3] relies on read-only tags inside the code segment, and numerous approaches use a table structure (made read-only) to store valid targets of indirect branches [97, 57, 58]. However, there are scenarios in which such page-level protection is unavailable, e.g., bare-metal systems which do not have a Memory Management Unit (MMU) and applications with dynamically generated code. Moreover, data race attacks [96], Rowhammer attacks [11] and Data-Oriented Programming (DOP) [40] have demonstrated that it is possible to gain arbitrary memory read and write access.

In this chapter, we explore the possibility of enforcing CFI in the absence of such an assumption. Specifically, we look into encoding CFI policies into the machine instructions directly without relying on policies specified in additional data structures (i.e., the read-only table structures in existing CFI approaches) or inserting CFI checks into the code segment. The general idea is to embed a statically constructed CFG to the instructions, execution of which is conditioned on correct control flows. In this way, each intended instruction will carry a proof that can validate the control-flow transfer. Unintended instructions cannot be executed as the proof in these instructions are not correct. Intuitively, instructions with CFG embedded can be seen as a proof-carrying code [56], where this proof is self-contained in the code rather than being encoded into a separate table. The challenge is how to embed the CFG into the instructions and how to correctly execute them at runtime.

Inspired by the framework of Instruction-Set Randomization (ISR) [42] where instructions of a program are encrypted with a secret key, we present Control-Flow Carrying Code ,C^3, which encrypts each basic block in the program with a key derived from the CFG. More specifically, the key is derived from (the addresses of) valid callers of the basic block to ensure correct control-flow transfers. At runtime, only the valid callers (their addresses) could enable the correct reconstruction of the key to decrypt the basic block. In this way, C^3 manages to embed and enforce CFI in the program instructions.

However, two challenges remain in making C^3 practical. First, a basic block may have multiple valid callers. These valid callers have different addresses, while the successor block has to be encrypted with a single key. How does C^3 enable the reconstruction of the single correct key by all the valid control-flow transfers? To address this challenge, C^3 utilizes the secret sharing scheme [75] to make the key shared among valid callers.

Although secret sharing helps solve this important challenge at a high level, we encounter more challenges in its application in our setting. For example, secret sharing requires that *all (a variable number of)* callers of the basic block be on the same secret sharing curve. The implication is that once we have the curve fixed, addresses of these callers can no longer take arbitrary locations but have to be on the secret sharing curve determined. This imposes extra challenges in laying basic blocks in the text segment of the program. To address this, we design an algorithm to redistribute basic blocks to positions satisfying the secret sharing curve.

We have implemented C^3 that consists of two components, one that performs binary rewriting to redistribute and encrypt basic blocks, and the other as a plug-in to an existing instrumentation platform to assist runtime execution of the rewritten executable. We apply C^3 to a number of server and non-server applications on the Linux platform. Our experimental results demonstrate that C^3 effectively defends against control-flow hijacking attacks and at the same time, introduces realistic runtime performance overhead for server applications comparable to existing Instruction-Set Randomization (ISR) implementations on the same instrumentation platform. Similar to the arguments in ISR systems, we believe that such overhead could be significantly reduced with a hardware-assisted platform.

4.2 Overview of C^3

4.2.1 Threat Model and Assumptions

The proposed defense, C^3, is aimed to protect a vulnerable application against control-flow hijacking attacks such as ROP attacks. The application to be protected may have some vulnerabilities that can be leveraged by an attacker to inject an exploit payload (code or data). We focus on user-space attacks leaving kernel exploits out of our scope. Specifically, we assume that:

- The target program does not contain self-modifying or dynamically-generated code.
- Attackers could use attacks to bypass W$\oplus$X, such as Data-Oriented Programming [40], data race [96] and Rowhammer attacks [11], and could exploit information disclosure vulnerabilities to investigate the victim's process memory.

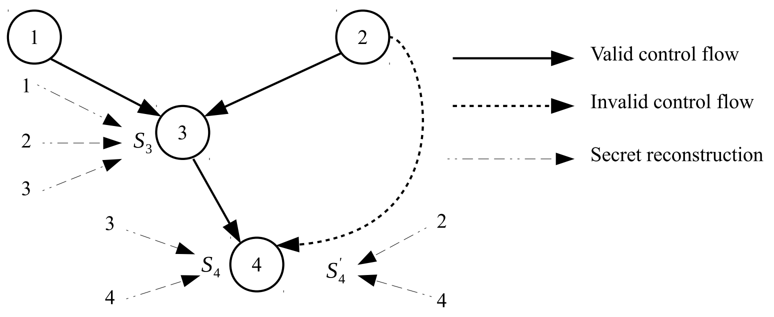


Figure 4.1: Example of secret reconstruction.

- Since the current implementation of C^3 is on top of the popular instrumentation platform Pin, we assume that attackers do not target Pin in their attacks and the partial memory segment managed by Pin (e.g., the code cache) is secure. This assumption can be removed if C^3 is supported by native hardware.

4.2.2 Embedding CFG to Instructions

Rather than consulting additional information stored in read-only memory, we propose to embed CFG to instructions. An instruction with CFG embedded can check the integrity of the control flow automatically during the execution without querying other data structures. In particular, C^3 embeds the CFG information by encrypting each basic block (an idea inspired by ISR [42]) with a key generated from control-flow dependent information. At runtime, the basic blocks are decrypted using a key reconstructed from the actual control-flow transfers taken. Only when the correct control flow paths are taken will the instructions be decrypted correctly.

In Figure 4.1, each node represents an encrypted basic block while edges indicate control flows. The solid edges represent valid control flows with S_i indicating the encryption key for basic block i. S_3 and S_4 are generated according to the valid control-flow path $< 1, 3 >$, $< 2, 3 >$ and $< 3, 4 >$. When there is an invalid control-flow transfer from node 2 to 4 denoted by the dotted edge, a wrong key S_4' is constructed which would result in illegal instruction faults.

Although the idea sounds straightforward, there are multiple design questions

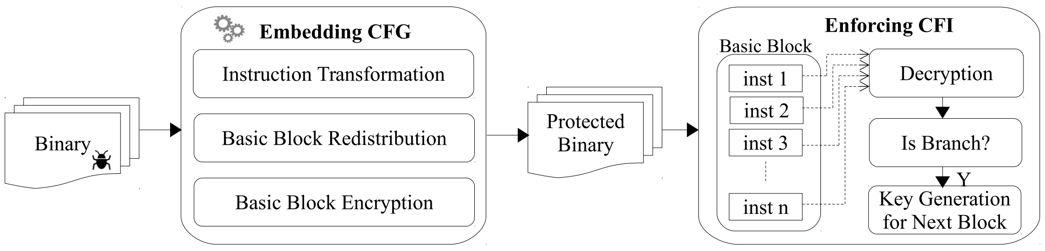


Figure 4.2: System overview of C^3.

and challenges. First, what information do we use to generate the key? Such information needs to be both statically and dynamically available, and it shall allow enforcement of CFI. How do we deal with basic blocks involved in multiple control flows, which may lead to different keys constructed dynamically? A simple solution is to insert (constant) shares at the caller and callee with which the secret can be reconstructed at runtime. However, such an approach does not provide Control-flow integrity because an attacker can reuse the share at other caller sites.

Our solution is to use the addresses of the branch transfer instruction and its target as the shares since they capture the control transfer information precisely. To deal with basic blocks involved in multiple control flows, we use basic block redistribution and secret sharing [75] to encode the key. Figure 4.2 shows an overview of C^3, consisting of two components.

- **Embedding CFG.** C^3 transforms branch transfer instructions (indirect branches, conditional jumps, and direct calls) to have a secret share embedded, and then redistributes basic blocks to specific addresses so that all valid callers are on the same secret sharing curve. Finally, basic blocks are encrypted with the secret.

- **Enforcing CFI.** Whenever the program attempts a control transfer, C^3 obtains the caller and callee addresses and reconstructs the key to decrypt the callee basic block before control transfer takes place.

4.3 Detailed Design of C^3

C^3 takes as input a binary executable (without source code) and outputs a modified executable with CFG embedded and CFI enforced.

4.3.1 Secret Sharing and Challenges

As discussed in Section 4.2, our approach of embedding CFG into instructions is to encrypt a basic block and to enable decryption with any correct control transfer. For a basic block with multiple callers, we can imagine that every valid caller shall contribute to the encryption key; however, in a concrete execution, only one valid caller is involved and the decryption key is reconstructed. This is where the idea of secret sharing comes to our design — only part of the ingredients of the secret key is needed for correct reconstruction. C^3 uses Shamir's approach [75] due to its simplicity.

The next question is the degree of the secret sharing equation. A general guideline is to keep it small to minimize overhead. We can use a degree of two with the source and target addresses of the control transfer — the minimum information to fully describe a control transfer. However, this runs into the risk of a code pointer disclosure exploit that discloses both addresses and allows an attack to decrypt the basic block. To counter such an attack, we add one random value (called the master key) which is unknown to the attacker to construct the secret key. Specifically, we use a degree of three, with the secret sharing equation

$$y = a_0 \ + \ a_1 x + \ a_2 x^2 \ (mod\, M) \tag{4.1}$$

where a_0 is the secret key for encryption and decryption, and x, y are k-bit coordinates extracted from the source and target addresses and the master key. C^3 obtains x and y from the lower-order odd- and even-index bits of an address (see Figure 4.3 for an example). Reconstruction of the secret follows Equation 4.2 with $x = 0$.

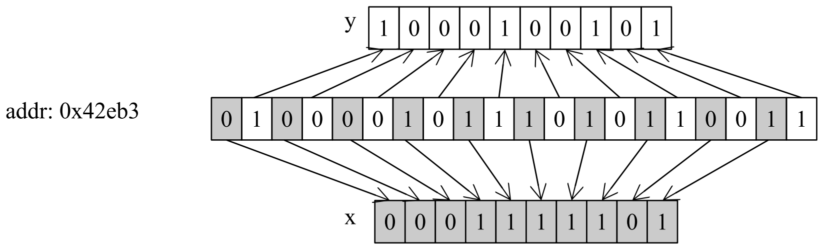


Figure 4.3: Extracting x and y for address 0x42eb3

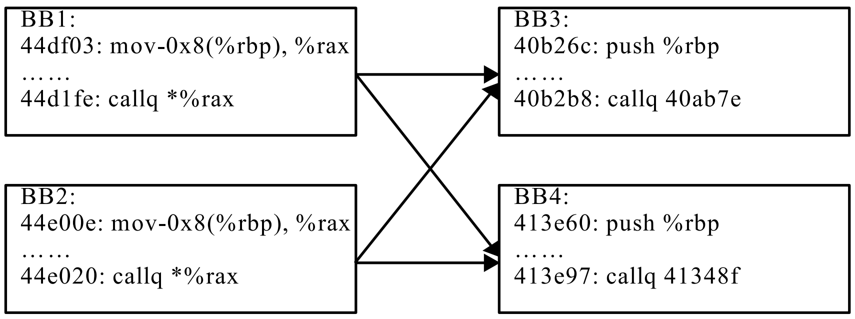


Figure 4.4: Multiple callers to multiple callees

$$y = \sum_{i=1}^{3} y_i \prod_{1 \le j \le 3, j \ne i} (x - x_j)(x_i - x_j)^{-1} \ (mod\ M) \tag{4.2}$$

To support a basic block with multiple callers, we can simply relocate the caller instructions so that they all lie on the parabola. However, a more challenging issue is to support a set of basic blocks with the same (set of) callers. Figure 4.4 shows an example with $BB3$ and $BB4$ having the same set of callers $BB1$ and $BB2$. Following the secret sharing design we outline above, the two parabolas for $BB3$ and $BB4$ will have three intersection points — the master key, $BB1$, and $BB2$; however, different parabolas could have up to two intersections only. Therefore, C^3 not only needs to relocate the basic blocks to move them onto specific parabolas, but also needs to perform some special transformations to control transfer instructions; see the next subsection.

4.3.2 Instruction Transformation

In fact, the challenge shown in Figure 4.4 is not the only one that C^3 needs to handle.

- **C1: Multiple callers to multiple callees.** In such cases, secret sharing curves for the callees have three or more intersections (including the master key), which is not possible for parabolas. We add an intermediate block between the callers and callees so that multiple callees now have a single caller.

- **C2: Basic blocks that are not freely movable.** Examples of such blocks include targets of `ret` instructions which must follow the `call` instruction, and the default branch of conditional instructions which must follow the conditional branch instruction. They cannot be moved freely to other locations due to the implicit control flow. Our strategy is to transform the implicit control flows into explicit ones.

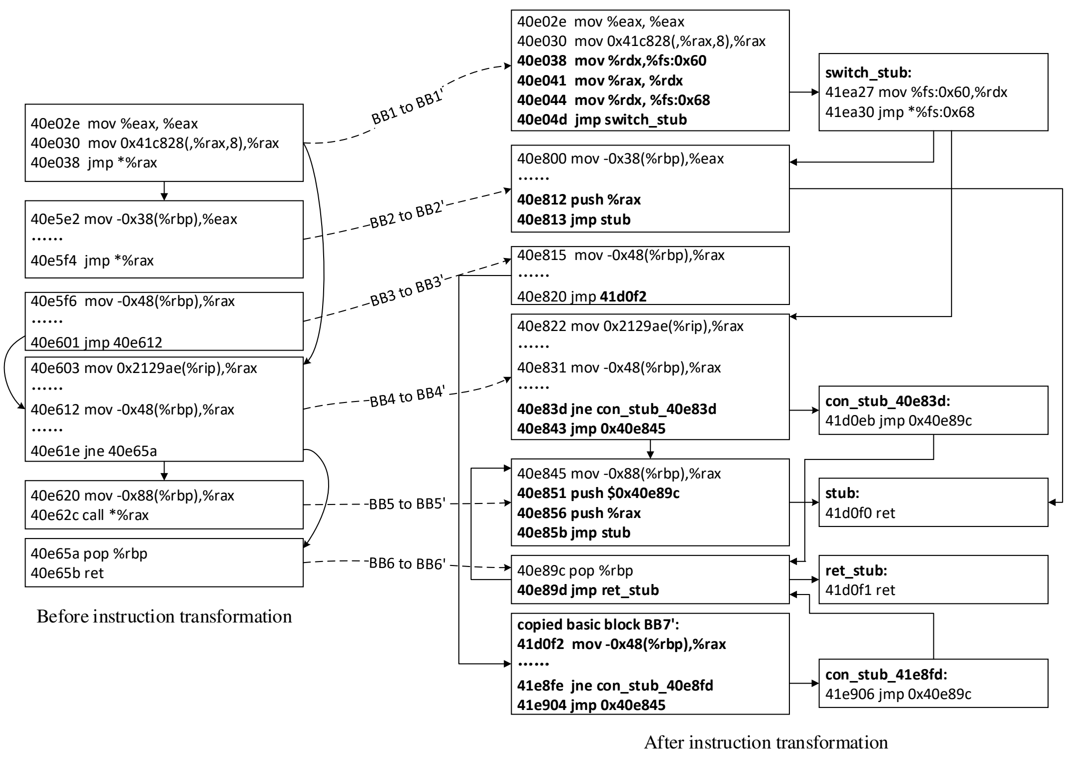
- **C3: Basic blocks with multiple entries.** Multiple entries will lead to different keys derived for the same basic block. Our strategy is to break it up into multiple basic blocks, each of which has a single entry.

In the rest of this subsection, we use an example (Figure 4.5) to explain how C^3 solves these complexities. Note that the transformation is via binary rewriting without source code of the program.

Transforming indirect call and indirect jump instructions

C^3 transforms an indirect call instruction into two `push` instructions (one to save the return address and the other to save the target address onto the stack) followed by a `jmp` instruction (jumping to a common stub); see $BB5$ and $BB5'$ in Figure 4.5. The stub block has a single `ret` instruction.

Although this simple transformation solves C1, it potentially enforces a relaxed CFI policy since multiple control transfer targets now go through the same common stub block. We stress that the same policy is used by existing coarse-grained CFI methods [94, 97]. Moreover, C^3 increases the difficulty of a stealthy attack since the valid targets are now encrypted. We could use a more complicated secret sharing curve to enforce a finer-grained policy, but C^3 chooses this solution due to its



```
40e02e  mov %eax, %eax
40e030  mov 0x41c828(,%rax,8),%rax
40e038  jmp *%rax

40e5e2 mov -0x38(%rbp),%eax
......
40e5f4  jmp *%rax

40e5f6  mov -0x48(%rbp),%rax
......
40e601 jmp 40e612

40e603 mov 0x2129ae(%rip),%rax
......
40e612 mov -0x48(%rbp),%rax
......
40e61e jne 40e65a

40e620 mov -0x88(%rbp),%rax
40e62c call *%rax

40e65a pop %rbp
40e65b ret
```

```
40e02e  mov %eax, %eax
40e030  mov 0x41c828(,%rax,8),%rax
40e038  mov %rdx,%fs:0x60
40e041  mov %rax, %rdx
40e044  mov %rdx, %fs:0x68
40e04d  jmp switch_stub

40e800 mov -0x38(%rbp),%eax
......
40e812 push %rax
40e813 jmp stub

40e815  mov -0x48(%rbp),%rax
......
40e820 jmp 41d0f2

40e822 mov 0x2129ae(%rip),%rax
......
40e831 mov -0x48(%rbp),%rax
......
40e83d jne con_stub_40e83d
40e843 jmp 0x40e845

40e845 mov -0x88(%rbp),%rax
40e851 push $0x40e89c
40e856 push %rax
40e85b jmp stub

40e89c pop %rbp
40e89d jmp ret_stub

copied basic block BB7':
41d0f2  mov -0x48(%rbp),%rax
......
41e8fe  jne con_stub_40e8fd
41e904 jmp 0x40e845
```

```
switch_stub:
41ea27 mov %fs:0x60,%rdx
41ea30 jmp *%fs:0x68

con_stub_40e83d:
41d0eb jmp 0x40e89c

stub:
41d0f0 ret

ret_stub:
41d0f1 ret

con_stub_41e8fd:
41e906 jmp 0x40e89c
```

Figure 4.5: An example of instruction transformation by C^3

simplicity and enforcing a CFI policy not less secure than existing work. Note that a byproduct of pushing the return address on the stack (the first `push` in $BB5'$) is a solution to C2, as the return site can now be freely moved (explained later in the next subsection).

Indirect jumps are handled in the same way, except that we only need one `push` instruction since there is not a return address, e.g., $BB2$ in Figure 4.5. Additional challenge arises here when the indirect jump was generated due to `switch/case` statements during compilation, where local variables are sometimes accessed via `%rbp` directly without changing `%rsp`. In such cases, we cannot simply push the target address of the indirect jump onto the stack because doing so would overwrite the local variables. Instead, we make use of thread local storage to store the target; see the indirect jump in $BB1$ of Figure 4.5. In order to transform an indirect jump `jmp *0x8(%rax)` (the target is the address in memory) while having the same switch stub with `jmp *(%rax)`, we simply move the target of them to the temporary register `%rdx` as shown in $BB1'$.

Transforming conditional jump instructions

Conditional jumps usually have a fall-through branch to the instruction that immediately follows, forming an implicit control transfer (C2). We turn this into an explicit one by inserting a direct jump instruction as in $BB4'$ of Figure 4.5. Note that similar to indirect jumps, conditional jumps may be followed by multiple callees (C1); that is why we also add a stub block as shown in $BB4'$ of Figure 4.5.

Transforming return instructions

Handling return instructions (C1) is simple as we only need to add a common stub which then returns to the call site; see $BB6'$ in Fig 4.5. We can enforce a finer-grained CFI policy by classifying functions into indirectly-called and directly-called ones, of which the latter does not need the additional stub block to be inserted since any two of them cannot return to the same call site. We leave this security improvement as our future work.

Transforming basic blocks with multiple entries

The multiple entries of a basic block correspond to different sets of ingredients for the secret reconstruction, and therefore will result in different keys (C3). C^3 handles this by copying each entry (and subsequent instructions in the block) to a new address and updating the corresponding control-flow instructions to the new addresses. For example, $BB4$ in Figure 4.5 has two entries, `0x40e603` and `0x40e612`, respectively. C^3 copies the second entry to a new address ($BB7'$) and directs the control flow from $BB3'$ to it.

4.3.3 Basic Block Redistribution

Redistributing basic blocks so that all callers of a control transfer are on the same secret sharing curve is an interesting and non-trivial problem. One can consider it as a directed graph traversal in which whenever a node is traversed, we pick a parabola and ensure that all its callers are on it by moving some or all the callers. However, if the traversal is not carefully designed, we could get into a failure where a node that has been previously moved on a parabola now needs to be moved again to satisfy another parabola — a mission impossible. Therefore, the key is to design a directed graph traversal algorithm that minimizes or eliminates such a risk.

C^3 uses a customized Depth First Search (DFS) algorithm. Intuitively, DFS fits our requirement in that it explores a branch to its ultimate leaves before backtracking or stepping into a new branch, which avoids unnecessary moving of caller nodes of branches already unexplored. We customize it with a "look ahead" capability which switches to another nearby branch when continuing exploring the current branch will get into a "mission impossible" case.

As shown in Figure 4.6 where shaded nodes denote those that had previously been moved (and therefore cannot be moved again) and hollow ones otherwise, continuing to traverse node A would run into a failure mode since node B will have two caller nodes fixed, making it impossible to find a parabola for node B (it already has three points determined including the master key). In this scenario, our "look ahead" function will traverse the sub-branch of node B before going back to traverse node A. This "look ahead" function is also used to decide the starting point. By default, C^3 picks a node with the largest number of callers as the starting point, and then uses the "look ahead" function to check whether this starting point and one of its callers target the same basic block. If they do, C^3 uses this basic block as the starting point. The detailed algorithm is shown in Algorithm 1.

Specifically, for a callee to be processed, we first check whether there is a prior basic block using the "look ahead" mechanism described above. Then, for each callee to be processed, we check whether there exists any of its callers that has a fixed address. If there is, we use this caller (with a fixed address) to determine the parabola; otherwise we randomly choose a caller to determine the parabola. The special and additional processing here is that for each (caller or callee) address, we need to check whether it will have the same x value with its callee, caller or the

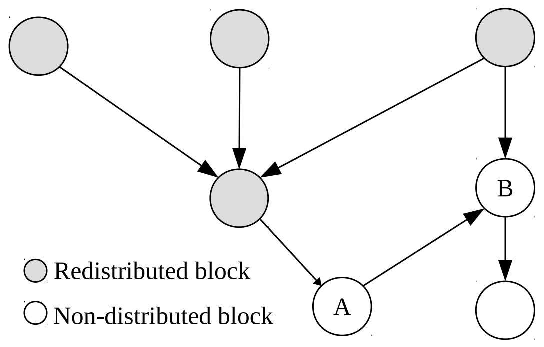


Figure 4.6: "Look ahead" DFS search

master key, since the same x value could result in a failure in calculating the inverse to compute the secret as in Equation 4.2. We generate a new random address if when detecting this problem.

Once a parabola is determined, we move all the callers onto it by randomly choosing an unused coordinate on the curve, which determines the new addresses of the callers. After that, we use the DFS approach to process other basic blocks.

Since the redistribution of basic blocks might turn a short jump instruction into a long jump, C^3 turns every direct jump into a long jump (with a four-byte displacement) before the redistribution process starts.

4.3.4 Encryption and Decryption

Before we present details of C^3 in encrypting a basic block, we note that completely separating code from data into different sections is a prerequisite for our encryption to work. This is because the encryption of any data may disrupt program execution when it is not decrypted at runtime. Fortunately, many linkers are configured to ensure such separation, and compiler optimizations like jump tables are also typically moved to a non-code section. C^3 does not include PLT calls in its protection as doing so will result in `.plt` section containing non-continuous addresses due to basic block redistribution (see the previous subsection), which in turn makes it impossible for the dynamic loader to update addresses in the Global Offset Table (GOT).

C^3 uses XOR as the encryption function due to its simplicity. The reconstructed

Algorithm 1 Basic Block Redistribution

```
1:  procedure REDISTRIBUTION(callee, master_key, k, p)
2:      if callee not in key_block then
3:          $priority\_callee = Look\_Ahead(callee)$
4:          if priority_callee then
5:              Redistribution$(priority\_callee, master\_key, k, p)$
6:          $callers$ = callee_caller[$callee$]
7:          $moved\_callers$ = find_moved_callers$(callers)$
8:          if len(moved_callers) == 0  then
9:              $caller$ = random_choose_caller$(callers)$
10:         if len(moved_callers) == 1 then
11:             $caller$ = moved_callers[0]
12:         compute_key$(callee, caller, master\_key, k, p)$
            ▷ % move all callers of this basic block to be on the curve %
13:         for i in callers do
14:             if i not in redistributed_block then
15:                 move_caller$(callee, i, master\_key, k, p)$
            ▷ % DFS: process callees of this basic block %
16:         for i in caller_callee[callee] do
17:             Redistribution$(i, master\_key, k, p)$
            ▷ % backtracking %
18:         for i in callers do
19:             for j in caller_callee[i] do
20:                 Redistribution$(j, master\_key, k, p)$
```

secret s from secret sharing is used as the seed to a pseudo-random function generator to generate a 16-bit key for encryption. The length of the secret s is a configurable parameter which has an upper bound of 16 because going beyond that may result in distance between two instructions greater than 2^{31}. To fight against memory disclosure attacks that attempt to compromise the master key, C^3 stores the master secret key outside of the binary into a database file, an approach used in some ISR approaches [65]. We note that C^3 could also perform load-time encryption on the basic blocks using a session key (replacing the master key) to further improve security [7, 60]. Also note that when the binary rewriting process is performed remotely, we could make use of remote attestation [23] to securely distribute the master key. We leave both ideas as our further work.

4.3.5 Transitioning from Unprotected to Protected Code

C^3 supports partial protection of a program that contains protected (CFG embedded) and unprotected (e.g., system or third-party libraries without CFG embedded) code. However, the transitioning from unprotected to protected code needs special attention since CFI checks will fail as the caller is not on the secret sharing curve of the callee. Such transitioning typically occurs in two scenarios.

- **Returning to protected code.** This happens when protected code calls an external library function and subsequently returns from it.
- **Calling to a function in protected code.** This happens when the external library function (e.g., qsort, bsearch) calls a comparison function in the protected code.

We handle these cases by adding a dummy block before each return target and function entry in the protected code, since we cannot accurately identify calls to a library function and functions called by the library. This dummy block has only one instruction that jumps to the actual target, and is encrypted with a key generated from its address. C^3 transfers control to the dummy block when detecting a control transfer from unprotected to protected code, the range of which is recorded into a (secure) database.

In this way, C^3 ensures that these dummy blocks cannot be invoked by control-flow transfers in the protected code and provides the same level of protection compared with existing CFI techniques.

4.4 Implementation

We implemented C^3 on an Ubuntu 64-bit system supporting inputs of ELF binary executables without source code.

4.4.1 Binary Rewriter

We developed our custom binary rewriter in 6,500 lines of Python code with the help of the disassembly engine Capstone [67]. The binary rewriter takes as input the ELF executable to be protected and the configuration of k. Embedding CFGs to an executable consists of three stages.

Before we embed control-flow information, we first obtain the static CFG. We do this by modifying a recent work typearmor [85] (which builds on Dyninst [8]).

Secondly, we use Capstone to disassemble the binary. C^3 uses the algorithm described in Algorithm 1 to select basic blocks and then compute the secret for each of them. Note that the redistribution algorithm will likely distribute basic blocks apart from each other, and many NOP instructions need to be inserted into the `.text` section.

In the last stage, we update the corresponding section information including program entry point, program header, section header, items in relocation table, `.dynamic`, and `.dynsym` sections. In addition, some instructions need to be updated to maintain the original control flow:

- **Direct jumps:** We transform all indirect branch transfers to jump to the stub first; see Section 4.3.2. Therefore, there are only direct jumps in the `.text` section now. The target address of a direct jump is specified as a relative offset from the address of the jump instruction, which needs to be recomputed after basic block redistribution.

- **PC-relative addressing mode:** We also need to patch instructions with PC-relative addressing mode, which are often used to generate position-independent code. The new x86-64 architecture natively supports PC-relative addressing, e.g., `lea ox200000 (%rip), %rbp` adds `0x200000` to the program counter and saves it to `%rbp`. To ensure correctness, C^3 updates these instructions by recomputing the new offset using the new program counter and the address of the redistributed target.

- **Function pointers:** They are usually absolute addresses of indirect call targets that are loaded into registers. To fix these instructions, the absolute address of the callee should be patched at the instruction that loads its address into the corresponding register. This is done by identifying all possible function pointers with the help of the symbol table and patching them to the redistributed addresses. The same goes to global function pointers where C^3 updates the address in data section.

- **Data pointers:** They need to be patched, too, because the starting offset of the data section has changed. C^3 patches them by adding the new offset to the original value.

- **Jump tables/virtual tables:** C^3 updates the base address of the jump table by adding the new offset to it. Patching virtual tables follows the same mechanism.

4.4.2 Execution Environment

We make use of Pin [50] to implement the execution environment with 1,100 lines of code in C++. It first reads from the secure database the master key and the protection range and then installs a callback that intercepts the loading of all images to obtain ranges of the unprotected memory. In addition to Pin, we can also make use of the dynamic code optimization platform called DynamoRIO [12] to implement the execution environment.

We then use the instrumentation callback at instruction granularity to detect a branch and compute the key for the next basic block. The decryption of basic blocks is performed by installing a callback that replaces Pin's default mechanism of fetching code from the target process. If the instruction fetched is within the range of protected code, we reconstruct the key from secret sharing parabola for decryption.

For code transitioning described in Section 4.3.5, we make use of `PIN_SetContextReg` to set the value of `%rip` register to the address of the dummy block which has just one instruction that jumps to the actual target, and then use the `PIN_ExecuteAt` API to direct execution to it. For the transition from protected code to unprotected code, C^3 stops the decryption and let the code execute as normal. Similar to other CFI approaches, the attacker can use gadgets in unprotected code to construct code-reuse attack, which C^3 cannot defend against.

To avoid performing frequent key reconstruction for direct branch transfer instructions, we cache the key for subsequent use. Therefore, each direct branch transfer instruction corresponds to only one key reconstruction.

4.5 Evaluation

We first analyze the security of C^3 and then measure its performance overhead with real-world applications.

4.5.1 Security

C^3 mitigates code-injection attacks in the same way Instruction-Set Randomization defeats them — when control flow is redirected to injected code, C^3 will decrypt it into random bytes. The attacker could not prepare the correct encrypted code since she does not know the master key.

C^3 also mitigates most Code-Reuse Attacks (CRA) due to three reasons. First, C^3 generates a wrong key when an invalid control transfer happens, which results in a random byte stream to be executed. Second, redistributing and encrypting basic blocks makes it harder for attackers to analyze and locate gadgets, which defeats most static CRA. Finally, the encrypted basic blocks result in little information revealed even when an attacker manages to dump the execution memory, which defeats most dynamic CRA.

Table 4.1: Comparison with existing CFI techniques.

Exploits	BinCFI [97]	CCFIR [94]	IFCC [84]	kBouncer [62]	ROPecker [19]	C^3
Göktas et al [36]	✓	✓				
Davi et al. [27]	✓			✓	✓	
Conti et al. [22]	✓	✓	✓			
Hu et al. [40]	✓	✓	✓	✓	✓	

Comparison with existing CFI techniques

A number of recent proof-of-concept exploits have shown how existing coarse-grained CFI techniques can be bypassed [36, 27, 22]. Although C^3 also enforces a coarse-grained policy, its unique handling of basic blocks (encryption) provides a new defense to make these exploits unsuccessful. Table 4.1 compares various CFI techniques with C^3 on the CFI policy enforced and defense capability against the exploits.

As shown in Table 4.1, existing instrumentation-based CFI methods [97, 94, 84] do not insert checks for unintended control-flow transfers, making them vulnerable to the exploit proposed by Conti et al. [22]. Such an exploit would not work on C^3 as all instructions (intended or unintended) are encrypted. The exploit proposed by Hu et al. [40] succeeds on all existing CFI methods as they rely on the assumption that $W \oplus X$ is effective. Moreover, the content in the CFI table inserted by BinCFI provides sufficient information about useful gadgets if there is memory disclosure. However, since C^3 does not have this problem because it does not insert any meta-data. The first three CFI approaches in Table 4.1 also suffer from TOCTOU attack — time of checking values of `esp/rsp` and time of executing `ret`, when the return address is stored in memory which could be modified by another thread. Under the protection of C^3, even if the address is modified by another thread, control flow will transfer to cipher-text which will result in program crashing.

Exploits that use call-preceded gadgets [36, 27] cannot succeed on C^3 since basic blocks are redistributed to random addresses. We performed experiments to verify the effectiveness of C^3 on defending against CRA that uses call-preceded gadgets using the test application `ndh_rop` from ROPgadget[1], a publicly available test set for ROP attacks. Our experiments verified that the payload that successfully exploits `ndh_rop` failed to run on C^3. Upon further investigation, we realized that it generated an illegal instruction fault when the return instruction directs control flow to the first call-preceded gadget. This is because this address is an invalid instruction which does not carry a valid proof to reconstruct the correct decryption key.

Compared with fine-grained approaches, e.g., Lockdown [63], which uses binary instrumentation to enforce CFI for different modules, C^3 can achieve better security as the attacker cannot make use of memory disclosure to traverse the memory of the victim program due to encryption of instructions. Basic block redistribution performed in C^3 can also be seen as effectively making the coarse-grained CFI policy finer-grained since the attacker cannot find the addresses of gadgets.

One may argue that the attacker could dump the protected code and do offline analysis to decrypt it. However, even if the attacker dumps the protected code and obtains the master key and the address of a basic block, she still has to try all possible encryption keys to see whether the basic block can be decrypted into valid instructions. We performed such experiments and realized that there are usually multiple such keys which have to be further tested on the resulting caller blocks for validity checks, and such checks have to carry on for callers of the callers, which makes it difficult for offline analysis to decrypt the protected code.

CFI effectiveness with AIR

Zhang and Sekar [97] propose using *Average Indirect target Reduction (AIR)* for measuring the strength of CFI, which has become a common method of evaluation [63, 87, 49]. It computes the average number of machine code instructions that are eliminated as possible targets of indirect control transfers.

[1]`https://github.com/JonathanSalwan/ROPgadget`

Table 4.2: Average indirect target reduction.

Programs	# of valid targets		AIR
	ends with indirect branch	ends with direct branch	
`vsftpd` ($k = 9$)	25	0	99.84%
`Pure-FTPd` ($k = 10$)	172	0	98.95%
`ProFTPD` ($k = 11$)	506	0	99.23%
`httpd` ($k = 11$)	171	0	99.74%
`Nginx` ($k = 11$)	125	0	99.81%
`lighttpd` ($k = 10$)	35	0	99.79%
`Memcached` ($k = 10$)	62	0	99.62%
average			99.57%

The formula used by Zhang and Sekar is shown in Equation 4.3, where n is the number of indirect branch instructions in the program, and S is the total number of instructions to which an indirect branch can transfer control flow, whose value is the same as the size of code in a binary. $|T_j|$ is the possible number of targets to which indirect branch j can transfer control flow after CFI enforcement.

$$\frac{1}{n}\sum_{j=1}^{n}(1 - \frac{|T_j|}{S}) \qquad (AIR) \qquad (4.3)$$

In the case of C^3, $|T_j|$ is the possible number of targets that can be interpreted as valid basic blocks for indirect branch j. Since we substantially increase the size of the `.text` section, instead of enumerating all possible addresses (which requires testing millions of addresses), we randomly choose $16,384$ addresses for effective testing when $k \leq 10$ and $65,536$ addresses for other k values. We consider all basic blocks ending with indirect transfer instructions as valid, and those ending with direct transfer instructions valid if their targets are in the `.text` section.

The results are shown in Table 4.2 with server applications. Interestingly, there are few addresses that can be interpreted as valid basic blocks, and all of them end with indirect transfer instructions. This is because C^3 extends the displacement in direct branches to four bytes, which makes the probability that a random sequence be interpreted as a valid direct branch small. On average, C^3 achieves an AIR value of 99.57%, comparable to existing CFI approaches [97, 63].

JIT-ROP

JIT-ROP [79] is an attack against fine-grained randomization. It assembles ROP gadgets "on-demand" without knowing the memory layout by exploiting the disclosure of a single code pointer. Specifically, the adversary traverses the memory space that the leaked pointer points to, searches for gadgets and cross-page transfer instructions to find new code pages and other useful gadgets. However, under C^3, a read performed from a code page yields cipher-text, which the adversary cannot disassemble without knowing the decryption key. As such, an adversary cannot use JIT-ROP to disclose new code pages to find gadgets.

To verify our intuition, we tried to use the ROP gadget finding tool peda[2] to identify gadgets in the protected binary `nginx-1.4.0` after the loading phase, simulating the full disclosure of the code segment. Many gadgets ending with `ret` are found, which are chained together to form an attack payload. However, the gadgets found were based on encrypted basic blocks, which become invalid instructions and lead the execution into an illegal instruction fault.

Blind ROP

Blind ROP [9] uses the response from the victim process (crash vs. no crash) as a side channel to incrementally guess the position of a gadget. It assumes that the adversary can disassemble the code pages to find the required gadgets. Since the code pages are encrypted with C^3, Blind ROP will not succeed. We applied the exploit script provided by Bittau et al.[3] to `nginx-1.4.0` protected by C^3, and found that it made all worker threads "stuck" as they were all running into an infinite loop of locating gadgets. Blind ROP uses a conservative implementation to incrementally populate the stack to find a stack-based stop gadget to avoid hanging. However, with C^3, every attempt in transferring control to this stack-based stop gadget results in a failure due to incorrect decryption of the callee block.

[2]`https://github.com/longld/peda`
[3]`http://www.scs.stanford.edu/brop/`

Control-Flow Bending

Control-Flow Bending (CFB) [15] bypasses conventional CFI that statically generates CFGs. CFB abuses certain functions whose executions may change their own return addresses to point to any call-preceded site which allows the attacker to "bend" the control flow. C^3 mitigates CFB attacks by preventing the attacker from locating call-preceded basic blocks — thanks to redistributing and encrypting of basic blocks.

Although C^3 successfully defends against these existing advanced control-flow hijacking attacks, we acknowledge that it is not necessarily effective against an attack specifically crafted for C^3. We further discuss this possibility in Section 4.6.

4.5.2 Performance overhead

We evaluated C^3 with three FTP servers (`vsftpd`, `ProFTPD`, and `Pure-FTPd`), three web servers (`Nginx`, `lighttpd`, and `Apache`), a distributed memory caching system (`Memcached`), and some common applications (image processing tools `sam2p`, `GraphicsMagic`, and `ImageMagics` and `bzip2`). All programs are executed with their default settings on a desktop computer with an Intel i7-4510u CPU with 8GB of memory running x64 version of Ubuntu.

To benchmark web servers, we configured Apache Benchmark[4] to issue 2,000 requests with 100 concurrent connections. For FTP servers, we configured pyftpbench benchmark[5] to open 20 connections and request 100 files per connection with over 100MB of files requested. To benchmark `Memcached`, we used memslap[6]. We ran each experiment 10 times, ensuring that the CPUs were fully loaded throughout the tests, and report the median.

Since C^3 is implemented on top of the dynamic instrumentation platform Pin, we measure the performance of C^3 in terms of the additional execution overhead com-

[4]`httpd.apache.org/docs/2.4/programs/ab.html`
[5]`http://code.google.com/p/pyftpdlib`
[6]`http://docs.libmemcached.org/bin/memslap.html`

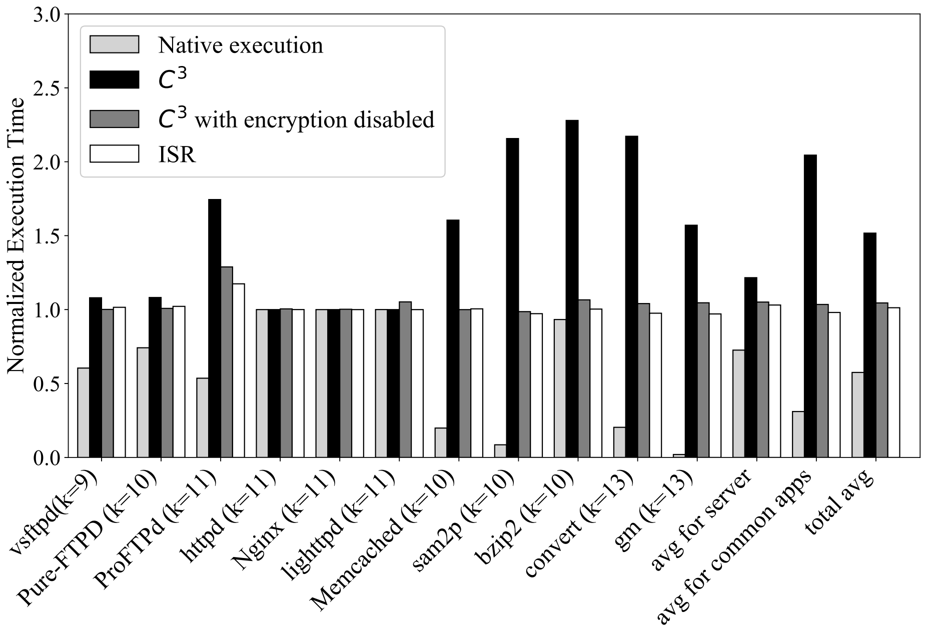


Figure 4.7: Overall overhead of C^3. The result is normalized to the baseline execution time on unmodified Pin.

pared to these programs executing on an unmodified Pin v.3.5. To enable a better understanding of the results, we also report the execution overhead of another system that is built on top of Pin, namely Instruction-Set Randomization implemented by Portokalidis et al. [65].

Execution Time

We report, in Figure 4.7, the execution time of each program under four settings: native execution, ISR [65], C^3 with encryption disabled, and C^3 with encryption turned on. Results are normalized to a baseline for its execution on unmodified Pin. k was chosen to be the minimum that successfully distributes the basic block for secret sharing, whose values are shown in brackets.

Being consistent with results reported in the original paper [65], ISR does not incur observable slow down compared with execution on unmodified Pin since there is no additional instrumentation. C^3 presents very similar results when encryption is disabled for the same reason. With encryption turned on, C^3 experiences less than 10% overhead for server applications while non-server applications generally suffer from significantly higher overhead. Note that when compared with their respective native executions, several server applications on C^3 have very small runtime performance, although the average runtime overhead is about 70%.

To gain a better understanding of contributions to such overhead and why non-server applications perform worse, we conduct the next finer-grained analysis of C^3 to see which components of C^3 are the main contributors to the overhead. We first identify the following three main tasks of C^3 that potentially contribute to the performance overhead:

- **Key Reconstruction (**KR**).** This is performed for every branch in the program, be it a direct branch (whose key reconstruction is denoted as dKR) or an indirect branch (whose key reconstruction is denoted as iKR).

- **Decryption.** Since C^3 uses XOR operation as in ISR [65], decryption incurs minimal overhead as confirmed in Figure 4.7 in which ISR only results in a small overhead.

- **Execution Redirection (**ER**).** This happens when execution transitions from unprotected code to protected code. Since it requires saving and restoring the entire register state [59], it could result in significant overhead.

Figure 4.8 shows the overhead of C^3 with certain components disabled to more accurately attribute the overhead to the corresponding components. This time, the overhead is presented in seconds without normalization (to visualize the small differences). We have two important observations.

First, iKR (whose contribution can be seen by comparing the bars for C^3 and those for C^3 with iKR disabled) contributes more overhead than dKR (whose contribution can be seen by comparing the bars for C^3 with iKR disabled and those for C^3 with KR disabled). This is mainly due to optimizations C^3 implements for direct branches, where key reconstruction is done only once and results are cached for subsequent decryption. Such optimization does not apply to indirect branches since

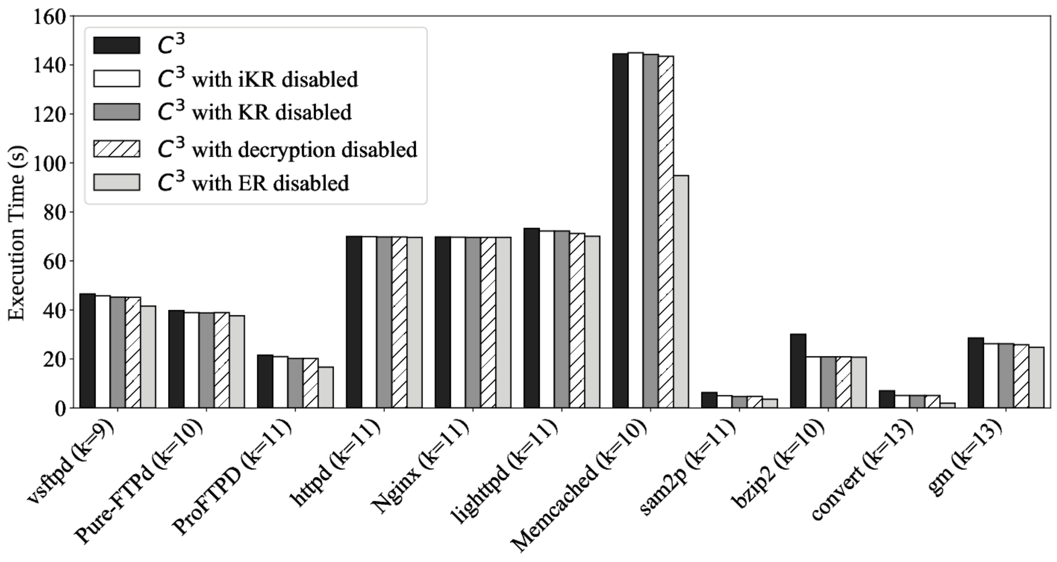


Figure 4.8: Detailed overhead of C^3.

Table 4.3: Number of various branches executed

Programs	iKR #	dKR #	ER #
`vsftpd`	3.99×10^6	2.43×10^7	1.38×10^6
`Pure-FTPd`	1.09×10^6	6.29×10^6	2.49×10^5
`ProFTPD`	5.07×10^6	5.67×10^7	1.68×10^6
`httpd`	1.16×10^5	4.82×10^5	9.97×10^4
`Nginx`	3.43×10^3	2.51×10^5	4.70×10^3
`lighttpd`	1.75×10^5	2.28×10^6	6.31×10^4
`Memcached`	2.37×10^7	1.54×10^8	7.64×10^6
`sam2p`	2.22×10^7	1.33×10^8	1.22×10^5
`bzip2`	1.36×10^8	7.64×10^9	5.83×10^4
`convert`	2.55×10^7	6.81×10^7	4.36×10^5
`gm`	1.14×10^6	4.25×10^7	1.21×10^5

the control transfer target changes in each indirect branch. Therefore, applications with more indirect branches suffer higher overhead on C^3.

Table 4.3 records the number of indirect branches, direct branches, and transitions from unprotected to protected code. Note that `bzip2` has a larger number of indirect branches executed, which explains its higher overhead on C^3.

Our second observation from Figure 4.8 is on execution redirection ER. We found that ER contributes significantly to the performance overhead for `vsftpd`, `proftpd`, `memcached` and the non-server applications except `bzip2`, which can be explained by the numbers in the last column of Table 4.3.

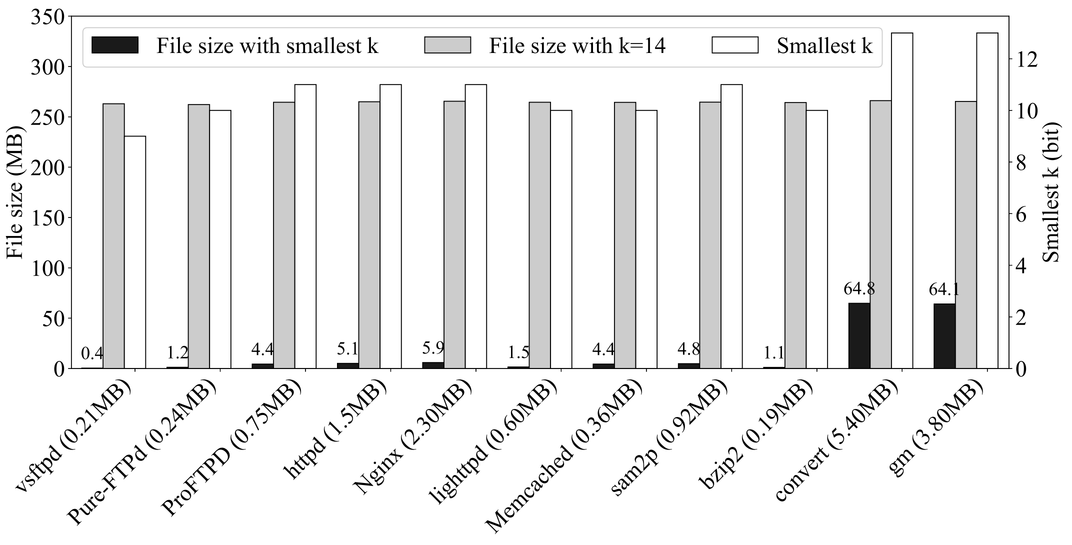


Figure 4.9: File size with different secret sizes.

Space Overhead

The redistribution of basic blocks in C^3 makes use of a potentially large address space with gaps among various basic blocks; see Section 4.3.3. The resulting size of the binary executable mainly depends on the length of the secret, i.e., k. For example when $k = 12$, the address of an instruction can be as big as 2^{24}.

Figure 4.9 shows the resulting file sizes after C^3 processing with two settings — one using a smallest possible setting of k (which varies among different programs) and the other with $k = 14$. We argue that although the size of the binary increases significantly with bigger values of k, storage is cheap and it is usually not an issue with hard-disk space. That said, a larger k also results in slightly bigger runtime overhead as more instructions are executed to extract the values of x and y from an address, and key reconstruction could also require slightly more instructions executed.

4.6 Discussion

4.6.1 Return-into-Pin

C^3 provides CFI protection on the application but not the dynamic instrumentation platform, i.e., Pin. An attacker, in theory, could perform an attack by returning into instructions in Pin so that control is diverted directly into the gadgets found in Pin. We call such an attack "return-into-Pin". Such a control transfer would circumvent C^3, enabling the attacker to successfully execute control-flow hijacking attacks. Our design of C^3 is compatible with other isolation hardening solutions, such as Software-based Fault Isolation (SFI) [86], though, which can instrument memory writes to check whether the application attempts to write to a page "owned" by Pin. Another (probably better) defense is to implement the execution environment in a more isolated layer such as the OS layer, the hypervisor layer, the hardware layer, or even inside SGX [23].

That said, once instructions are in Pin's code cache, Pin will not instrument them but jump there directly, which improves the performance of C^3. Meanwhile, such optimization does not hurt security since Pin uses a local hash table for each individual indirect branch transfer, which will contain only the correctly decrypted targets. Any new targets will result in a hash table miss and basic block decryption.

4.6.2 Return-into-libc

In general, CFI does not defend against all return-into-libc attacks. Specifically, C^3 does not encrypt instruction sequences in the `.plt` section, and so any return instructions can transfer control to entries in the `.plt` section. In order to protect these library function calls, one could statically compile the libraries into the application.

4.6.3 Length of the Keys

Brute-force attacks have been introduced to reconstructing the encryption keys in ISR [81], which is also applicable to C^3. However, since we use a different key for encrypting each basic block, such brute-forcing will be ineffective because a successful attack typically requires the reconstruction of keys for multiple basic blocks. To this end, we believe that using XOR as the encryption algorithm for improved performance is justifiable, although C^3 can definitely use a more secure encryption scheme. We currently use a 32-bit master key since it is unique for the entire program. C^3 could improve its security with a longer master key of, say, 80 bits.

4.6.4 Fine-grained CFI Enforcement

C^3 can be extended to enforce fine-grained CFI. For example, C^3 can enforce the fine-grained CFI policy for forward-edge indirect branch transfer instructions with our improved policy described in Section 3.1.4 by classifying functions and indirect call instructions into different clusters according to the number of arguments they can accept, and then encrypting basic blocks with the more accurate set of control transfers derived. We note that enforcing a finer-grained CFI policy could likely reduce the execution time and space overhead of C^3 due to fewer valid control transfers on average and consequently less secret sharing and block redistribution needed.

4.6.5 Other limitations

First, C^3 relies on static analysis and rewriting of binaries. The current implementation does not support dynamically generated code or self-modifying code.

Second, C^3 prevents attackers from directly reading the code and finding useful gadgets. However, code pointers in data areas such as stack and heap are still vulnerable to indirect memory disclosure. For example, if the protected binary has a format string vulnerability, the attacker can print out the valid memory locations for return instructions, which may allow an attacker to use, e.g., call-preceded gadgets. This is a rather general limitation shared by other techniques performing binary rewriting [97, 94, 87].

Third, C^3 renders caching and pipelining less effective. It is a limitation for most ISR approaches, excluding those performing decryption when there are I-cache misses and store plain text in the I-cache.

Lastly, C^3 requires symbol names in the executable to enable patching function and data pointers after basic block redistribution. It also requires that data and code be completely separated to enforce instruction encryption. For binaries that do not contain symbol information, we can use external tools, e.g., Unstrip[7] and others [66, 77], to restore the symbol information. Similarly, there are approaches to identify data embedded within code [95, 97].

4.7 Summary

We present C^3, a new CFI technique that embeds the CFG into instructions to perform CFI checks without relying on additional data structure like the read-only table used in existing CFI approaches. It encrypts each basic block with a key that can be reconstructed by any of its valid callers with the help of a secret sharing scheme. During execution, C^3 reconstructs the key when a branch transfer instruction is encountered. Our evaluation shows that C^3 can effectively defend against most control-flow hijacking attacks with moderate overhead.

The content of this chapter is based on our previously published work [47] with no major changes.

[7] `http://paradyn.org/html/tools/unstrip.html`

Chapter 5

Control-Flow Integrity Enforcement with Dynamic Code Optimization

5.1 Introduction

Prior to the introduction of CFI in 2005, there have already been a lot of research on dynamic code optimization to improve performance of dynamic program interpreters. For example, Wiggins/Redstone [32], Dynamo [5], Mojo [18], and DynamoRIO [12]. Although most of these were not proposed by the security community, there is at least one noticeable work called *program shepherding* [43] which makes use of a general purpose dynamic optimizer RIO [12] to enforce security policies. DynamoRIO and program shepherding provide nice interfaces for enforcing security policies on control transfers, which makes us believe that they can be good candidate architectures for CFI enforcement. Since these well established and mature dynamic code optimizers are proven to introduce minimal overhead, we believe that they could result in a system that significantly outperforms existing CFI implementations.

In this chapter, we present *DynCFI* that enforces a set of security policies on top of DynamoRIO for CFI properties. We detail how this set of policies are designed and implemented, and show that *DynCFI* achieves similar security properties when compared to a number of existing CFI implementations while experiencing a much lower performance overhead of 14.8% as opposed to 28.6% of *BinCFI*. We stress that *DynCFI* is not necessarily an CFI enforcement implementation that has the lowest performance overhead. Instead, our contribution lies on the utilization of the dynamic code optimization system which is a matured system proposed and well studied before CFI was even introduced, and to the best of our knowledge, *DynCFI* is the first implementation of CFI enforcement on top of a dynamic code optimizer.

In the second half of this chapter, we further investigate the exact contribution to this performance improvement. We propose a three-dimensional design space and perform comprehensive experiments to evaluate the contribution of each axis in the design space in terms of performance overhead. Among many interesting findings, we show that traces in the dynamic optimizer, which consist of cached basic blocks stitched together, had contributed the most performance improvement. Results show that traces have decreased the performance overhead from 22.7% to 14.8%. We also evaluate how branch prediction and indirect branch lookup have changed the performance. To the best of our knowledge, this is the first comprehensive evaluation on the performance overhead contributed by various components of the system, and we believe that this detailed understanding would aid future research and development of efficient CFI enforcement systems.

5.2 Design, Implementation, and Security Comparison

Our objective is to design a practical and efficient CFI enforcement without the extra requirement of recompilation or dependency on debug information. In this section, we first present the design of DynCFI that can be effectively enforced on DynamoRIO and the implementation of it, and then compare the security property it achieves with some existing CFI (and related defense) approaches. Before intro-

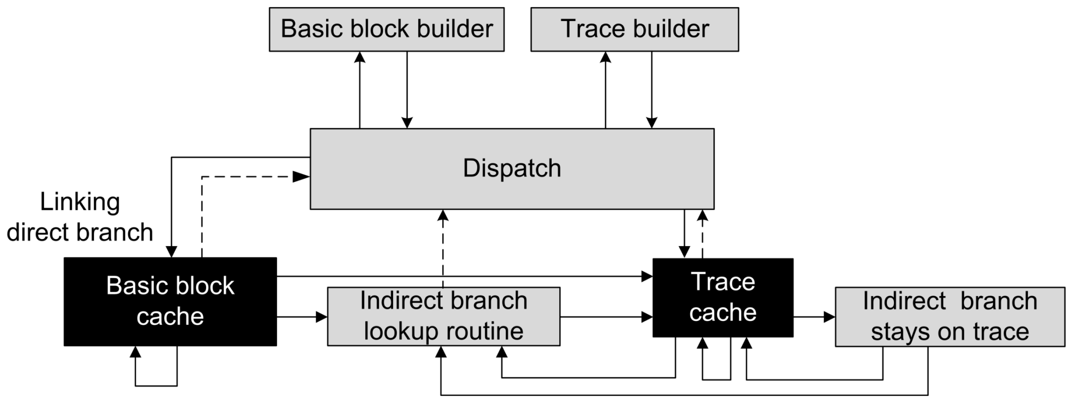


Figure 5.1: Overview of *DynamoRIO*

ducing the details of *DynCFI*, we first overview the workflow of DynamoRIO.

5.2.1 DynamoRIO

Figure 5.1 shows an overview of *DynamoRIO* [12], with darker shading indicating the application code to be monitored.

DynamoRIO first copies basic blocks into the basic block cache. If a target basic block is present in the code cache and is targeted via a direct branch, *DynamoRIO* links the two blocks together with a direct jump. If the basic block is targeted via an indirect branch, *DynamoRIO* goes to the indirect branch lookup routine to translate its target address to the code cache address. Basic blocks that are frequently executed in a sequence are stitched together into the trace cache. When connecting beyond a basic block that ends in an indirect branch, a check is inserted to ensure that the actual target of the branch will stay on the trace. If the check fails, it will go to the indirect branch lookup routine to find the translated address.

To make itself a secure platform on which programs are executed, *DynamoRIO* splits the user-space address into two modes: the untrusted application mode and the trusted and protected RIO mode. This design protects *DynamoRIO* against memory corruption attacks. Meanwhile, the beauty of *DynamoRIO* (and the corresponding good performance) come mainly from the indirect branch lookup which is very efficient in determining control transfer targets with a hashtable. This hashtable

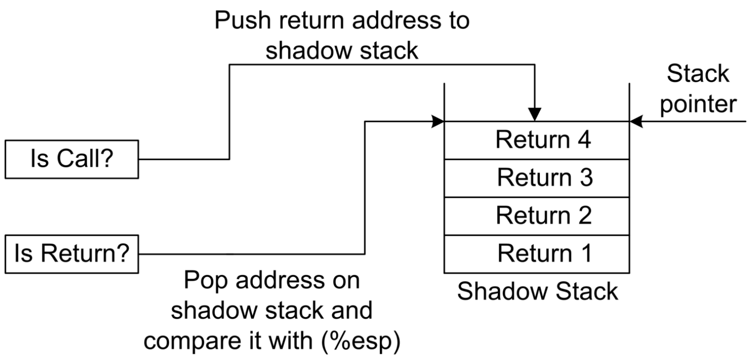

Push return address to shadow stack
Stack pointer
Is Call?
Return 4
Return 3
Return 2
Is Return?
Return 1
Pop address on shadow stack and compare it with (%esp)
Shadow Stack

Figure 5.2: Shadow stack operations

maps the original target addresses with addresses in the basic block cache and trace cache so that most control transfers require minimal processing.

5.2.2 Returns

The most frequently executed indirect control transfer instructions are returns. DynCFI maintains a shadow call stack for each thread to remember caller information and the corresponding return address. The whole process is shown in Figure 5.2. For a call instruction, we store the return address on our shadow stack. For a return instruction, we check whether the address on the shadow stack equals to the address stored at the stack memory specified by %esp. Such a shadow stack enables DynCFI to apply a strict policy that only returning to the caller is allowed, although a relaxed version could also be applied to reduce overhead.

5.2.3 Indirect jumps and indirect calls

We further classify indirect jumps into normal indirect jumps and PLT jumps, such as `jmp offset (base_register)`, which are used to call functions in other modules, target of which can only be exported symbols from other modules. To obtain target information for every indirect branch, we use the static analysis engine provided by another well-known CFI enforcement *BinCFI* [97], which combines linear and recursive disassembling techniques and uses static analysis results to ensure correct disassembling. Targets of indirect calls are function entry points and targets of indirect jumps are function entry points and targets of returns. Meanwhile, targets of PLT jumps are exported symbol address. These valid jump and call targets are organized into three different hashtables to improve performance—one for indirect jumps, one for indirect calls, and one for PLT jumps.

Most importantly, the shadow stack and hashtables we used can just be readable and writable in the DynamoRIO mode, in the user mode, they are readable only, so attackers cannot modify their contents.

5.2.4 Implementation

As discussed in Section 5.2.1, the indirect branch lookup routine in *DynamoRIO* maintains a hashtable that maps original control transfer target addresses with addresses of code caches. The hashtable has to be built when the control transfer occurs the first time though. This process, together with the dispatcher which is invoked when matches are not found in the hashtable (see Figure 5.1), become the natural place of our CFI enforcement, since CFI mainly concerns control transfer targets.

We obtained the source code of *DynamoRIO* version 5.0.0 from the developer's website [1] and added about 700 lines of code (in C) to implement *DynCFI*. Most of the additional code is added to the dispatcher where checks of control-flow transfers are performed. Some code is also added to basic block cache building to implement our shadow call stack and to initialize *DynamoRIO* to load the valid jump/call target addresses into our own hashtables.

DynCFI does not implement the full sets of CFI properties originally proposed by Abadi et al. [3]. In particular, we only perform checks on indirect control transfers at the first time when the target of an indirect branch occurs. However, it does not really impact security, and it is exactly the reason why *DynamoRIO* is widely accepted as an efficient dynamic optimizer — original code is cached in short se-

[1]`http://www.dynamorio.org/`

Table 5.1: Security comparison with other CFI and ROP defenses

Approach	Policy			
	Return	Indirect jump	Indirect call	PLT jump
BinCFI [97]	Call-preceded	Function entry,return address	Function entry	Exported symbol address
CCFIR [94]	Corresponding springboard section			Nil
CFIMon [90]	Call-preceded	Any address in the training set	Any function entry	Nil
ROPdefender [28]	Caller	Nil	Nil	Nil
kBouncer [62]	Call-preceded	Nil	Nil	Nil
LockDown [63]	Caller	Function entry, instruction in the current function	Function entry	Nil
DynCFI	First execution: Caller, Others: Call-preceded	Function entry,return address	Function entry	Exported symbol address

quences and security policies, if any, need only be checked the first time the code cache is executed [12]. Subsequent executions of the same code cache will be allowed (without checking) as long as the control transfer targets remain unchanged. Any violations to our policy will miss the (very efficient) indirect branch hashtable lookup and go back to the dynamic interpreter which will consider the control transfer a first timer and perform all the checks (inefficient).

5.2.5 Security comparison

DynCFI provides comparable security properties with most existing CFI implementation and ROP defense solutions. Table 5.1 shows *DynCFI* (last row) when compared to some of these other approaches.

A caveat here is that in order to improve performance, we make use of the shadow call stack information only when a new target is added to the hashtable (i.e., not checking the shadow call stack if the target address is found in the hashtable). This will make the policy effectively call-proceeded only. Since call-proceeded policy is widely considered as adequate by many other approaches, we apply this performance improvement in our subsequent evaluation. This relaxed policy also enables a fair comparison between *DynCFI* and other CFI enforcement schemes since many others also use a call-proceeded policy.

DynCFI achieves similar security when compared with these existing approaches. In particular, *DynCFI* is mostly comparable to *BinCFI* in that both maintain a list of valid target addresses to be checked at runtime, with one noticeable difference in the enforcement mechanism: *BinCFI* enforces the policies with static instrumentation to translate indirect target address while *DynCFI* uses *DynamoRIO* as the interpreter platform. This makes *BinCFI* the perfect candidate for performance overhead comparison with *DynCFI*, which is the topic of our next Section.

5.3 Detailed Performance Profiling

In this section, we conduct a comprehensive set of experiments on the performance overhead of *DynCFI*. Besides the overall performance overhead, we run some detailed performance profiling to find out the contribution to such overhead by various components of the dynamic optimizer. We wish that such a detailed profiling could shed light on the part that contributes most to the performance overhead, and give guidance to future research in further improvement.

To better understand our evaluation strategy, we present our first attempt in the profiling, show the results, and explain the limitation of this attempt. We then choose an existing CFI implementation for the detailed comparison with *DynCFI*. We analyze the design space of CFI enforcement implementation and organize it along three axes on which the two systems under comparison could be clearly identified. Lastly, we perform a sequence of experiments by modifying individual components of *DynCFI* so that the contribution of each to performance overhead can be evaluated.

5.3.1 Target applications

To evaluate the performance overhead, we need to subject *DynCFI* (and another CFI implementation for comparison purposes) to some applications. To enable fair

Table 5.2: Percentage of time spent on various components

Application	Application code	IBL inlined	IBL not inlined	BB building	Trace building	Dispatch	Others
bzip2	97.99	0.60	0.00	0.20	1.20	0.00	0.00
gcc	86.78	7.46	0.26	0.91	3.42	1.10	0.07
mcf	97.48	0.42	1.26	0.14	0.07	0.14	0.49
gobmk	80.00	1.08	0.00	2.70	11.35	4.86	0.00
sjeng	94.10	5.67	0.11	0.02	0.09	0.02	0.00
libquantum	99.51	0.49	0.00	0.00	0.00	0.00	0.00
omnetpp	84.88	14.50	0.38	0.06	0.15	0.03	0.01
astar	94.36	4.79	0.78	0.00	0.01	0.04	0.01
namd	99.89	0.69	0.00	0.00	0.02	0.00	0.00
soplex	74.21	25.42	0.03	0.10	0.10	0.10	0.02
povray	89.71	6.88	0.82	0.76	1.01	0.76	0.06
lbm	99.99	0.00	0.00	0.00	0.01	0.00	0.00
Average	91.57	5.62	0.30	0.41	1.45	0.59	0.06

comparison with existing work, we used twelve pure C/C++ programs we can find in SPEC CPU2006, which are also used in the evaluation of the original work of *BinCFI* [97], as our benchmarking suite.

Experiments were executed on a desktop computer with an i7 4510u CPU and 8GB of memory running x86 version of Ubuntu 12.04. Each individual experiment was conducted 10 times, average of which is reported in this paper.

5.3.2 First attempt in performance profiling

As an initial attempt to understand the performance overhead contributed by various components of *DynCFI*, we use program counter sampling to record the amount of time spent in various components of *DynCFI*. We use the `ITIMER_VIRTUAL` timer which counts down only when the process is executing and delivers a signal when it expires. The handler used for this signal records the program counter of the process at the time the signal is delivered. We sample the program counter every ten milliseconds.

Table 5.2 shows the percentage of time each application spends in various steps in *DynCFI*. It suggests that more than 90% of the time is spent on the application's code on average. Other non-negligible processes include Indirect Branch Lookup (IBL) inlined with the application's code and that not inlined, basic block and trace

Table 5.3: Statistics of different types of control transfers

Application	%Indirect call	%Indirect jump	%Return	%Direct branch	Total (10^8)
`bzip2`	0.002	0.002	0.774	99.222	28
`gcc`	0.434	1.958	7.767	89.841	407
`mcf`	0.001	0.029	5.402	94.568	50
`gobmk`	0.001	0.027	4.811	95.161	7
`sjeng`	1.072	2.289	4.718	91.921	1229
`libquantum`	0.000	0.000	0.242	99.758	7068
`omnetpp`	1.609	1.763	33.998	62.630	875
`astar`	1.698	0.049	19.738	78.515	306
`namd`	0.000	0.008	3.292	96.700	1159
`soplex`	0.002	0.018	23.239	76.741	731
`povray`	2.776	0.154	26.279	70.791	81
`lbm`	0.000	0.017	0.035	99.948	152

cache building, as well as the dispatcher.

In an attempt to explain why some applications, e.g., `gcc`, `omnetpp`, `soplex`, and `povray`, incur larger overhead, we count the number of different control transfers in each application (runtime) and present statistics in Table 5.3. The correlation between the two tables suggests that larger number of control transfers could lead to the higher overhead.

Although it sounds like we have obtained detailed understanding of the performance overhead, there is one important factor that we have overlooked so far—the overhead contribution of the dynamic optimizer on executing the application's code (second column of Table 5.2). In other words, Table 5.2 does not tell us if the dynamic optimizer had sped up or slowed down the execution of the application's code, and what had contributed to that speedup or slowdown. Our further comparison verifies this suspicion, see Table 5.4, as there is noticeable difference in the amount of time spent.

Therefore, we want to further investigate the contribution of various components of the dynamic optimizer in speeding up or slowing down the application's code. We present our second attempt in the rest of this section.

With the objective of finding out contributions to the performance overhead by individual components of the dynamic optimizer, our strategy is to

1. Find an existing CFI implementation X for comparison.

Table 5.4: Time spent in application code

Application	in *DynCFI* (sec)	Natively (sec)	Overhead (%)
`bzip2`	4.88	4.86	0.41
`gcc`	60.73	56.25	7.96
`mcf`	13.91	14.19	-1.97
`gobmk`	1.48	1.35	9.62
`sjeng`	158.93	150.01	5.95
`libquantum`	813.12	821.63	-1.04
`omnetpp`	138.54	122.23	13.34
`astar`	76.16	75.44	0.95
`namd`	735.51	733.73	0.24
`soplex`	64.81	61.15	5.98
`povray`	14.21	14.12	0.64
`lbm`	375.45	388.14	-3.27

2. Continuously disable or modify individual components of C^3 so that the modified system eventually becomes similar to the implementation of X.

3. In every step of disabling or modifying the components, perform experiments to find the corresponding (difference in) performance overhead.

5.3.3 Picking BinCFI for detailed comparison

With this strategy, it is important that we choose an X that

- Is an independent, state-of-the-art implementation of CFI enforcement;

- Shares the same high-level idea with *DynCFI* while validating control transfers with a different approach (e.g., by binary instrumentation) from that of the dynamic optimizer/interpreter as in *DynCFI*.

so that our evaluation could attribute the difference in performance overhead to the dynamic optimizer.

BinCFI and *DynCFI* are similar in that both maintain a set of valid control transfer targets and use a centralized validation routine for CFI enforcement. In both cases, the validation routine maintains a hashtable for the valid control transfer targets.

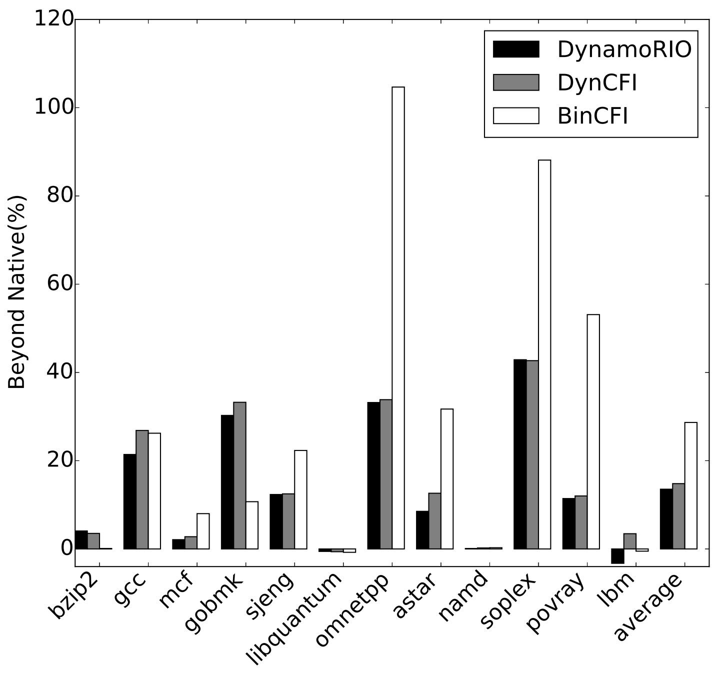


Figure 5.3: Overall performance overhead

The difference between *BinCFI* and *DynCFI* is that *BinCFI* obtains the valid target addresses of indirect branches statically and records their corresponding instrumented target addresses into the hashtable, and then replaces the indirect instructions with a direct jump to the CFI validation routine. *BinCFI* satisfies our requirements for the performance comparison, and is therefore chosen for our subsequent detailed evaluation.

5.3.4 Overall comparison and the design space

The overall performance overhead of executing the benchmarking applications under (original, unmodified) *DynamoRIO*, *DynCFI*, and (original, unmodified) *BinCFI* is shown in Figure 5.3. Results are shown in terms of percentage overhead beyond natively executing the applications on an unmodified Linux Ubuntu system. We obtained the source code implementation of *BinCFI* [97] from its authors.

An interesting observation is that the original *DynamoRIO* and *DynCFI* do not differ much in terms of overhead (a relatively small 1.3% difference). This shows that the interfaces provided by *DynamoRIO* are convenient and effective for CFI enforcement, which confirms our intuition since *DynamoRIO* intercepts all control transfers and no additional intercepting is needed in our modification to *DynamoRIO*.

DynCFI experiences a significantly smaller overhead of 14.8% compared to *BinCFI* at 28.6%. This suggests that the dynamic optimizer provides a more efficient platform for CFI enforcement compared to existing approaches like binary instrumentation as in *BinCFI*. That said, the two systems differ in other aspects and therefore this overall evaluation result is insufficient in attributing the majority of the performance gain to mechanisms of the dynamic optimizer.

As discussed in Section 5.3.3, our strategy to this difficulty is to continuously disable or modify individual components of C^3 so that eventually it becomes similar to *BinCFI*, in terms of their operating mechanism as well as the performance overhead. By doing so, we would likely observe degradation of performance (increase in overhead) of the modified system which is definitely due to the corresponding feature disabled or modified. The question is – which individual component or feature to disable or modified?

To answer this question, we analyze the internal validation mechanisms of the two approaches and identify three main factors that could significantly contribute to the different performance overhead.

1. **Trace** Trace is the most important mechanism in *DynamoRIO* to speed up indirect transfers. Traces are formed by stitching together basic blocks that are frequently executed in a sequence. Benefits include avoiding indirect branch lookups by inlining a popular target of an indirect branch into a trace (with a check to ensure that the target stays on the trace and otherwise fall back to the full security check), eliminating inter-block branches, and helping branch prediction. Trace is unique in *DynamoRIO* and is not in *BinCFI*.

2. **Branch prediction** Modern processors maintain buffers for branch prediction, e.g., Branch Target Buffer (BTB) and Return Stack Buffer (RSB). The

effectiveness of these predictors could get seriously affected due to the modifications to the control transfers. For example, turning a return instruction into a indirect jump would make RSB useless in the branch prediction, potentially leading to an increase in the performance overhead.

3. **Indirect branch lookup routine** Besides implementation details that are not necessarily due to the architectural design (to be discussed more in Section 5.3.5), a dynamic optimizer could use a single lookup routine for the entire application including the dynamically loaded libraries, while systems that apply static analysis and binary instrumentation would likely have to use a dedicated lookup routine for each module because some dynamically loaded libraries might not have been statically analyzed or instrumented. This could contribute to noticeable differences in performance overhead.

We want to explore details into these three axes to see how each of them affects the performance overhead. Other factors that might contribute to the overhead in C^3 which we do not further investigate include

- Building basic block caches;
- Building trace caches;
- Inserting new entries into hashtables;
- Context switches between DynamoRIO and code caches.

5.3.5 Profiling along the three axes

With identification of the three axes, we make our second attempt in detailed understanding of the performance overhead of the two systems. Since executing on *DynCFI* and executing on the original unmodified *DynamoRIO* experience about the same overhead (see Figure 5.3), our subsequent experiments will only focus on

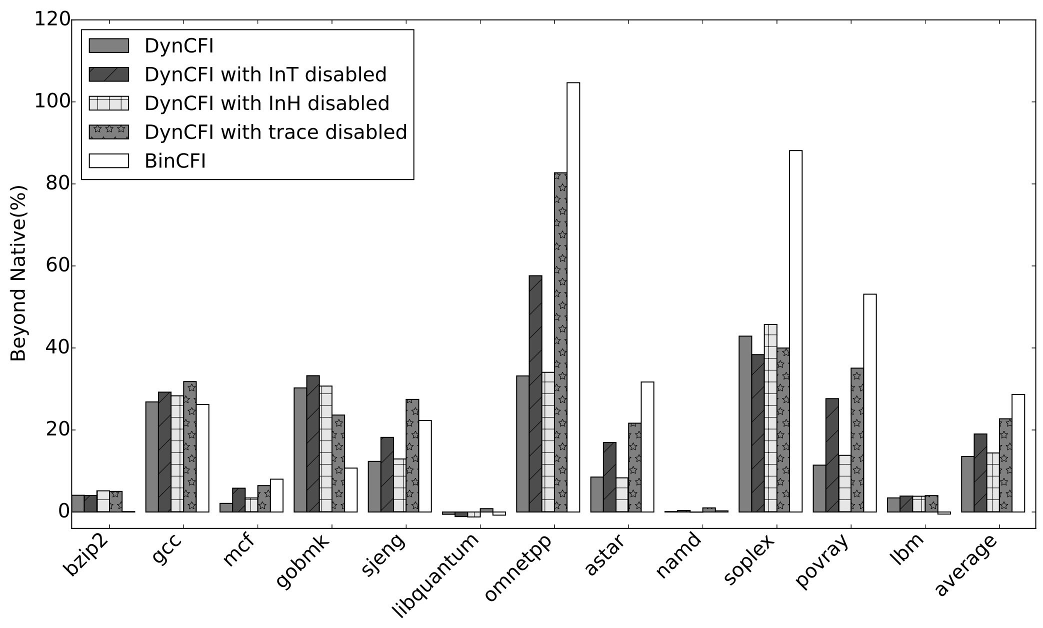


Figure 5.4: Impact of trace on overhead

comparing *DynCFI* and *BinCFI*. Also recall that our strategy is to disable or modify one component of *DynCFI* at a time and observe the corresponding change in performance overhead.

Traces

Traces are unique in dynamic optimizers like *DynamoRIO* and *DynCFI*. There are potentially two ways in which traces impact the performance overhead. First, the stitching of basic blocks together eliminates some inter-block branches. Second, each trace has inlined code to check if the control transfer target is still on the trace (we call this InT). If the target is still on the trace, execution will just carry on without further checking; otherwise, a second inlined code (we call this InH) is executed to perform hashtable lookup without collisions. If collision happens, execution will go to the full indirect branch lookup routine (denoted as R). We examine contribution of InT and InH by disabling them individually. We also examine the effect of traces overall and present the results in Figure 5.4.

Figure 5.4 shows that the contribution due to InT is big, averaging to 5.5%. Exceptions go to `bzip2` and `soplex` which do not gain much with InT mainly

Table 5.5: Execution of indirect control transfers

Original transfer		Return	Indirect call/jump
C^3	Basic block cache	Jump to R, indirect jump to target	
	Trace cache	InT or InH or jump to R, indirect jump to target	
BinCFI		Return	jump to R, indirect jump to target

because the fall-back of InH is very effective on them (which can be verified from the next-to-zero time spent in IBL not inlined in Table 5.2).

Although performance overhead increases when disabling InT (see Figure 5.4), *DynCFI* is still better than *BinCFI*. When disabling traces altogether, the overhead of *DynCFI* increases from 14.8% to 22.7% on average, with some going over the overhead in *BinCFI*. This shows that traces are contributing significantly in the low overhead of *DynCFI*. For applications with a large percentage of indirect branches (see Table 5.3), *DynCFI* with traces disabled still outperforms *BinCFI*. This suggests that there are other contributing factors in *DynCFI* which we have not evaluated.

Branch Prediction

The way in which *DynCFI* and *BinCFI* intercept and deliver control flow transfers has an implicit effect on branch prediction. Branch prediction is typically achieved by remembering a history of control transfer targets by the same instruction. Both *DynCFI* and *BinCFI* could weaken branch prediction due to R using the same instruction (an indirect jump) to execute control transfers originally executed by different instructions in the application [12, 97]. Table 5.5 summaries how indirect control transfers in an application are executed in *DynCFI* and *BinCFI*.

In summary, *DynCFI* leads *BinCFI* in retaining branch prediction for indirect calls and jumps when trace caches are used due to InT and InH; however, *BinCFI* would perform better than *DynCFI* for returns. That said, note that there are typically far more return instructions than indirect calls and jumps executed for all the applications in our benchmarking suite, see Table 5.3.

To better understand the effect of various components of *DynCFI* and *BinCFI*

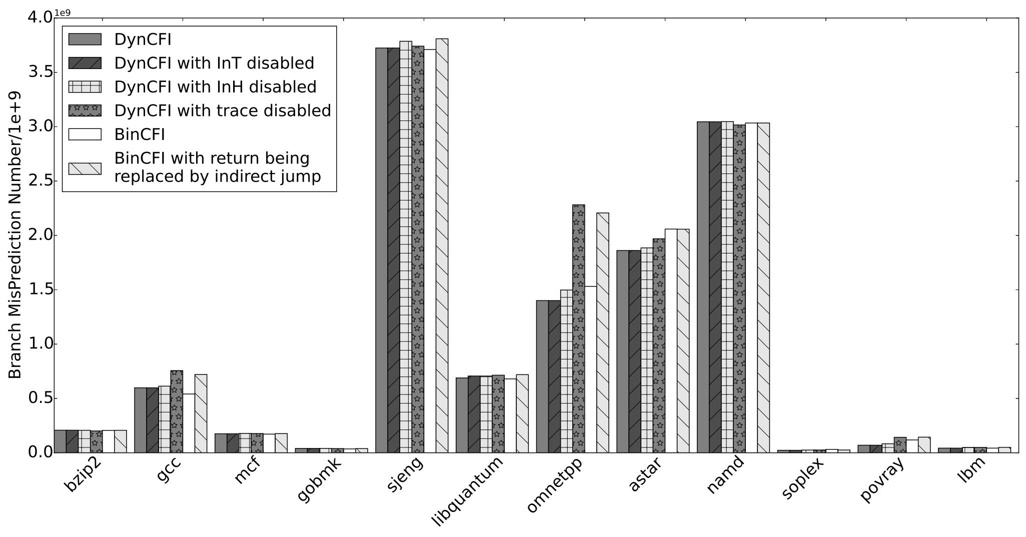


Figure 5.5: Impact of traces on the number of branch mispredictions

on branch prediction, we count the number of mispredictions when executing the benchmarking applications on a number of different settings – *DynCFI*, *DynCFI* with InT disabled, *DynCFI* with InH disabled, *DynCFI* with traces disabled, *BinCFI*, *BinCFI* with returns being replaced by jumps to R, and present the results in Figure 5.5.

We observe that disabling InH has a larger impact on branch prediction than disabling InT in general. This shows that the inlined hashtable lookup has its fair share of its contribution on lower overhead. It also indirectly shows that the hashtable implementation in *DynCFI* is good in that collisions do not happen often (since R not inlined is not executed often as shown in Table 5.2). Another interesting finding is that replacing returns with indirect jumps on *BinCFI* adds large number of mispredictions for some programs. In terms of overhead, this translates to about 2% more in the overhead as shown in Figure 5.6.

Indirect branch lookup routine R

The indirect branch lookup routine in *DynCFI* and *BinCFI* very much shares the same strategy. Both use an efficient implementation of a hashtable to record valid control transfer targets. One noticeable difference, though, is that *BinCFI* requires an extra step to check if the target resides within the same software module before

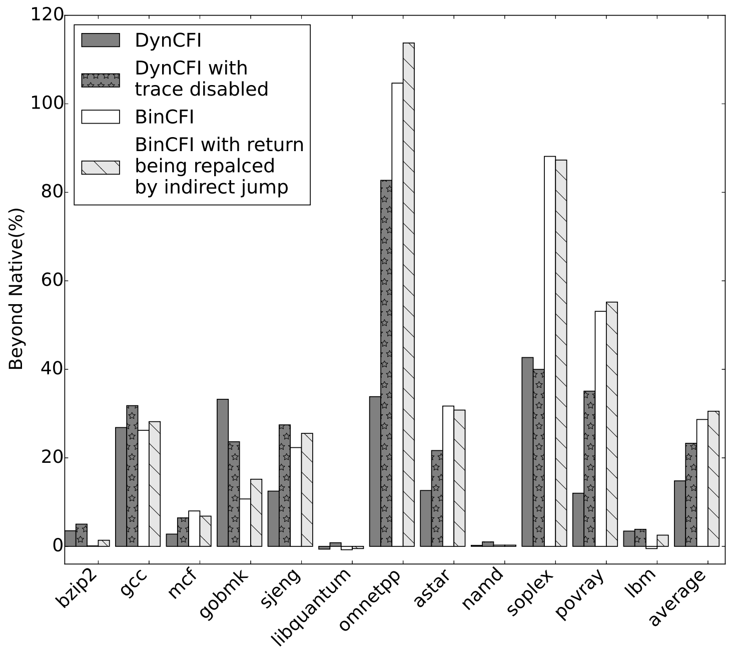


Figure 5.6: Impact of branch prediction on overhead

directing control to the corresponding R. Each software module has to implement its own copy of R because some dynamically loaded libraries might not have been statically analyzed or instrumented and *BinCFI* cannot use a centralized R for all modules.

On the other hand, *DynCFI* executes the application on top of a dynamic interpreter without static analysis or binary instrumentation, and therefore has three centralized R (one for returns, one for indirect jumps, and one for indirect calls) for all software modules. This architectural difference contributes to some additional performance overhead to *BinCFI*.

Besides the difference due to the architectural design, there are also lower level differences in implementing R between *DynCFI* (inheriting the same R from *DynamoRIO*) and *BinCFI*. In particular, they differ in the indirect jump instructions used (*DynCFI* uses a register to specify the target while *BinCFI* uses a memory), the number of registers used throughout the algorithm (and as a result the number of registers to be saved and restored), and efficiency of the hashtable lookup algorithm.

To evaluate the contribution of R in the overall performance overhead, we re-

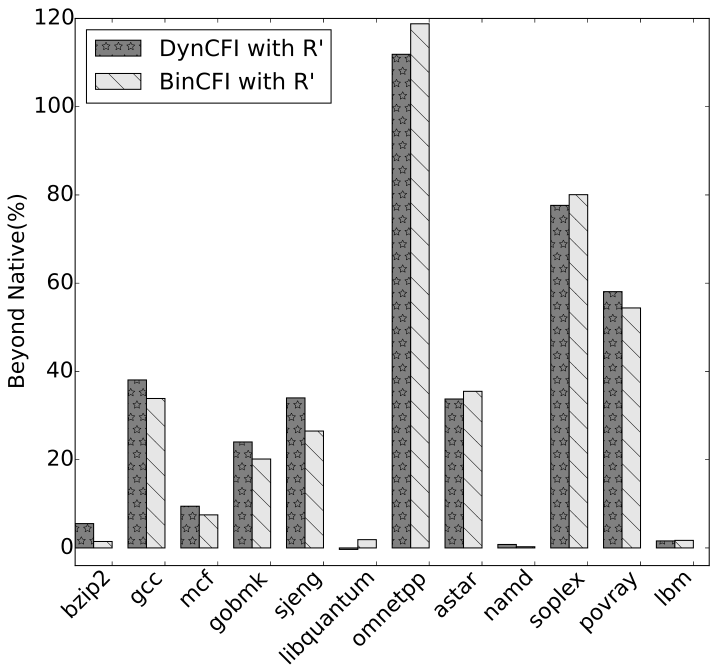


Figure 5.7: Performance overhead with unified R′

place R in both *DynCFI* (with traces disabled) and *BinCFI* (with returned replaced with indirect jumps) with R′, our (supposedly more efficient) implementation of the algorithm, and show the resulting performance overhead in Figure 5.7.

Comparing these results with those shown in Figure 5.6, we find that such low-level details in the implementation of R translates to significant differences in the overhead. In particular, the difference between C^3 and *BinCFI* shrinks with R′ replacing R, indicating that the original R used in *DynCFI* is more efficient than that in *BinCFI*.

5.4 Security evaluation and Discussion

5.4.1 Real world exploits

We use a publicly available intrusion prevention evaluator RIPE [89] to verify that *DynCFI* offers comparable security properties with existing CFI proposals (as analysis presented in Section 5.2.5). In particular, we check if *DynCFI* can detect exploits that employ the advanced Return-Oriented Programming (ROP) techniques.

Table 5.6: AIR metrics for SPEC CPU 2006

Name	DynCFI(%)	BinCFI(%)
`bzip2`	99.95	99.37
`gcc`	97.60	98.34
`mcf`	98.58	99.25
`gobmk`	98.18	99.20
`sjeng`	99.60	99.10
`libquantum`	98.10	98.89
`omnetpp`	99.61	97.68
`astar`	96.70	98.95
`namd`	99.99	99.59
`soplex`	99.49	98.86
`povray`	99.19	98.67
`lbm`	98.56	99.46
Average	98.80	98.86

RIPE contains 140 return-to-libc exploits out of which 60 exploit return instructions and 80 exploit indirect call instructions. For the 60 exploits on return instructions, our experiments confirm that *DynCFI* manages to detect all of them because they violate the call-preceded policy we enforced on return instructions.

DynCFI and *BinCFI* share the weakness in detecting exploits that change the value of a function pointer to a valid entry point of a function. Such attacks cannot be detected by most other CFI implementations either [90].

RIPE also contains 10 ROP attacks using return instructions, which are all successfully detected by *DynCFI* as the targets of these gadgets are not call-preceded.

5.4.2 Average indirect target reduction

Zhang and Sekar [97] propose a metric for measuring the strength of CFI called Average Indirect target Reduction (AIR). As *DynCFI* uses different policy on return branches, we apply the same metric to test *DynCFI* when applied to the SPEC benchmarking suite. Table 5.6 compares the AIR metrics for *DynCFI* and *BinCFI*. We can find that average AIR for *DynCFI* is 98.80% which is comparable to 98.86% for the case of *BinCFI*.

5.4.3 Fine-grained CFI Enforcement

DynCFI can be extended to enforce fine-grained CFI. For example, *DynCFI* can enforce the fine-grained CFI policy for forward-edge indirect branch transfer instructions with our improved policy described in Section 3.1.4 by classifying functions into different clusters according to the number of arguments they can accept, and then store them into different lookup tables.

5.5 Summary

In this chapter, we present *DynCFI*, a new implementation of CFI properties on top of a well-studied dynamic code optimization platform. We show that *DynCFI* achieves comparable CFI security properties with many existing CFI proposals while enjoying much lower performance overhead of 14.8% on average compared to that of a state-of-the-art CFI implementation *BinCFI* at 28.6%. Our detailed profiling of *DynCFI* shows that traces, a mechanism in the dynamic code optimization platform, contribute the most to such performance improvement.

The content of this chapter is based on our previously published work [49] with no major changes.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This dissertation makes contributions in recovering, embedding, and enforcing control-flow integrity policies.

In the first work presented in Chapter 3, we systematically study how compiler optimization would impact function signature recovery with 1,344 real-world applications with various optimization levels, and propose a novel improved mechanism to more accurately recover function signatures. The results show that compiler optimizations have both positive and negative impacts on function signature recovery. The second part of Chapter 3 studies the possible complication cases that aren't handled by the current deep learning approach and propose our approach by including domain-specific knowledge to the dataset.

In the second work presented in Chapter 4, we propose C^3, a novel approach to embed CFI policies into instructions rather than consulting read-only tables or inserting tags into the code section compared to other approaches. It embeds the CFI policies and its enforcement into instructions of the program by encrypting each basic block with a key derived from the control-flow graph. This embedded CFI policies can come from the improved mechanism described in Chapter 3. More specifically, we can classify functions and indirect call instructions into different clusters according to the number of arguments they can accept, and then encrypting basic blocks with the more accurate set of control transfers derived. This kind of "proof-carrying" code ensures only valid control-flow transfers can decrypt the corresponding instruction sequence and any unintended control-flow transfer would cause program crash. The security evaluation shows that C^3 is able to defend against most control-flow hijacking attacks while suffering from moderate runtime overhead.

The third work presented in Chapter 5 presents *DynCFI*, an efficient way to enforce CFI based on the dynamic code optimization platform DynamoRIO. The result shows that *DynCFI* enjoys much lower performance overhead of 14.8% on average compared to that of a state-of-the-art CFI implementation BinCFI at 28.6%. We further perform comprehensive evaluations and shed light on the exact amount of savings contributed by the various components of the dynamic optimizer including basic block cache, trace cache, branch prediction, and indirect branch loopup. We can make use of the CFI policies described in Chapter 3 to make our approach more fine-grained by classifying functions into different clusters according to the number of arguments they can accept, and then store them into different lookup tables. Moreover, the execution environment used in C^3 can also be replaced by DynamoRIO to improve performance.

6.2 Future Work

Given this dissertation, it will be interesting and valuable to further work on the following research directions:

1. *Control-Flow Carrying Code for Dynamically Generated Code.* As discussed in Section 4.6.5, C^3 does not support dynamically generated code. Due to the nature of these code, they have to be both writeable and executable. Although some defenses [80] are proposed to protect these dynamically generated code,

the attacker can still construct attacks. Since C^3 doesn't require the support of DEP, it is natural to apply it to the dynamically generated code.

2. *Correctness of Decompilation When Compiler Optimization enabled.* Decompilation that converts an executable into source code has been widely used in many areas, including malware analysis [91] and off-the-shelf software security hardening [24, 29, 30]. As discussed in the dissertation, compilers do not preserve much source-level information in the process of compilation, especially when optimization is enabled. However, the research community does not have an up-to-date understanding of the correctness of the decompilation output, which may impede reaching the full potential of modern decompilation tools in conducting research. In the future, we plan to study how compiler optimization would impact the correctness of decompilation.

Bibliography

[1] The heartbleed bug. `http://heartbleed.com/`.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX symposium on operating systems design and implementation*, pages 265–283, 2016.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[4] S. Andersen and V. Abella. Data execution prevention. *Changes to functionality in microsoft windows xp service pack*, 2, 2004.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[6] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 13rd International conference on compiler construction*, pages 5–23. Springer, 2004.

[7] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289. ACM, 2003.

[8] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16. ACM, 2011.

[9] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[10] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[11] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *Proceedings of the 37th IEEE symposium on security and privacy*, pages 987–1004. IEEE, 2016.

[12] D. Bruening. *Efficient,Transparent,and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

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

[14] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. Technical report, California Univ Berkeley Dept of Electrical Engineering and Computer Science, 2009.

[15] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, pages 161–176, 2015.

[16] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, pages 385–399, 2014.

[17] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[18] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A Dynamic Optimization System. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.

[19] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and H. Deng. others. 2014. ropecker: A generic and practical approach for defending against rop attack. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium.*

[20] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Security Symposium*, pages 99–116, 2017.

[21] D. D. I. F. Committee et al. Dwarf debugging information format, version 4. *Free Standards Group*, 2010.

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

[23] I. Corporation. Intel software guard extensions (intel sgx). `https://software.intel.com/en-us/sgx/`, 2019.

[24] N. Corteggiani, G. Camurati, and A. Francillon. Inception: System-wide security testing of real-world embedded systems software. In *Proceedings of the 27th USENIX Security Symposium*, pages 309–326, 2018.

[25] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[26] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 243–255, 2015.

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

[28] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[29] Y. David, N. Partush, and E. Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 392–404, 2018.

[30] Y. David and E. Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 349–360, 2014.

[31] R. de Clercq, J. Götzfried, D. Übler, P. Maene, and I. Verbauwhede. Sofia: Software and control flow integrity architecture. *Computers & Security*, 68:16–35, 2017.

[32] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Redstone: An On-line Program Specializer. In *Proceedings of the IEEE Hot Chips XI Conference*, 1999.

[33] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 51–60, 2013.

[34] J. Fu, X. Zhang, and Y. Lin. An instruction-set randomization using length-preserving permutation. In *Proceedings of the 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 376–383. IEEE, 2015.

[35] Ghidra. The ghidra decompiler. `https://ghidra-sre.org/`, 2019.

[36] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 575–589. IEEE, 2014.

[37] P. Guide. Intel® 64 and ia-32 architectures software developer's manual. 2016.

[38] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the 33th IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

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

[40] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, pages 969–986. IEEE, 2016.

[41] I. INTEL. Intel® 64 and ia-32 architectures software developer's manual. 2018.

[42] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[43] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11st USENIX Security Symposium*, volume 92, 2002.

[44] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd international symposium on Code generation and optimization*. IEEE Computer Society, 2004.

[45] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Network and Distributed System Security Symposium*, 2011.

[46] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes. Taming undefined behavior in llvm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 633–647. ACM, 2017.

[47] Y. Lin, X. Cheng, and D. Gao. Control-flow carrying code. In *Proceedings of the 14th ACM Asia Conference on Computer and Communications Security*, pages 3–14, 2019.

[48] Y. Lin and D. Gao. When function signature recovery meets compiler optimization. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, 2021.

[49] Y. Lin, X. Tang, D. Gao, and J. Fu. Control flow integrity enforcement with dynamic code optimization. In *Proceedings of the 19th International Conference on Information Security*, pages 366–385. Springer, 2016.

[50] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 26th ACM conference on Programming language design and implementation*, pages 190–200. ACM, 2005.

[51] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 941–951. ACM, 2015.

[52] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2014.

[53] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR abs/1301.3781*, 2013.

[54] A. Milburn, H. Bos, and C. Giuffrida. SafeInit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *Proceedings of the 24th Network and Distributed System Security Symposium*, pages 1–15, 2017.

[55] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. τ cfi: Type-assisted control flow integrity for x86-64 binaries. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 423–444. Springer, 2018.

[56] G. C. Necula. Proof-carrying code. design and implementation. In *Proof and system-reliability*, pages 261–288. Springer, 2002.

[57] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 577–587. ACM, 2014.

[58] B. Niu and G. Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 914–926. ACM, 2015.

[59] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk. Controlling program execution through binary instrumentation. *ACM SIGARCH Computer Architecture News*, 33(5):45–50, 2005.

[60] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis. Asist: architectural support for instruction set randomization. In *Proceedings of the 20th ACM conference on Computer and communications security*, pages 981–992. ACM, 2013.

[61] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33th IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[62] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent {ROP} exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, pages 447–462, 2013.

[63] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

[64] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *Proceedings of the 25th IEEE Symposium on Security and Privacy*, 2(4):20–27, 2004.

[65] G. Portokalidis and A. D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 41–48. ACM, 2010.

[66] R. Qiao and R. Sekar. Function interface analysis: A principled approach for function recognition in cots binaries. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 201–212, 2017.

[67] N. A. Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 2014.

[68] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib (c). In *Proceedings of the 25th Annual Computer Security Applications Conference*, pages 60–69. IEEE, 2009.

[69] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the 20th International Symposium on Software Testing and Analysis*, pages 100–110. ACM, 2011.

[70] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 21–28. ACM, 2010.

[71] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[72] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-rop defenses. In *Proceedings of the 17th International Workshop on Recent Advances in Intrusion Detection*, pages 88–108. Springer, 2014.

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

[74] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[75] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[76] A. Sharma, Y. Tian, and D. Lo. Nirmal: Automatic identification of software relevant tweets leveraging language model. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 449–458. IEEE, 2015.

[77] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Security Symposium*, pages 611–626, 2015.

[78] K. Sinha, V. P. Kemerlis, and S. Sethumadhavan. Reviving instruction set randomization. In *Proceedings of the 10th International Symposium on Hardware Oriented Security and Trust*, pages 21–28. IEEE, 2017.

[79] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[80] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *Proceedings of the 22nd Network and Distributed System Security Symposium*, 2015.

[81] A. N. Sovarel, D. Evans, and N. Paul. Where's the feeb? the effectiveness of instruction set randomization. In *Proceedings of the 15th USENIX Security Symposium*, 2005.

[82] D. Sullivan, O. Arias, D. Gens, L. Davi, A.-R. Sadeghi, and Y. Jin. Execution integrity with in-place encryption. *arXiv preprint arXiv:1703.02698*, 2017.

[83] P. Team. Pax address space layout randomization. *http://pax. grsecurity. net/docs/aslr. txt*, 2003.

[84] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in {GCC} & {LLVM}. In *Proceedings of the 23rd USENIX Security Symposium*, pages 941–955, 2014.

[85] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, pages 934–953. IEEE, 2016.

[86] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. 27(5):203–216, 1994.

[87] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 331–340. ACM, 2015.

[88] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[89] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime Intrusion Prevention Evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 41–50. ACM, 2011.

[90] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12. IEEE, 2012.

[91] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, pages 158–177. IEEE, 2016.

[92] Y. Yang. Rops are for the 99 `https://cansecwest.com/slides/2014/ROPsareforthe99CanSecWest2014.pdf`, 2014.

[93] D. Zeng and G. Tan. From debugging-information based binary-level type inference to cfg generation. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*, pages 366–376. ACM, 2018.

[94] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.

[95] M. Zhang, M. Polychronakis, and R. Sekar. Protecting cots binaries from disclosure-guided code reuse attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 128–140, 2017.

[96] M. Zhang and R. Sekar. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 91–100.

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