Static Binary Instrumentation with Applications to COTS Software Security

A Dissertation presented

by

Mingwei Zhang

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2015

Stony Brook University

The Graduate School

Mingwei Zhang

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

R. Sekar - Dissertation Advisor
Professor in Department of Computer Science

Mike Ferdman - Chairperson of Defense
Assistant Professor in Department of Computer Science

Michalis Polychronakis - Committee Member
Assistant Professor in Department of Computer Science

Zhiqiang Lin - Committee Member
Assistant Professor in Department of Computer Science
University of Texas at Dallas

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Static Binary Instrumentation with Applications to COTS Software Security

by

Mingwei Zhang

Doctor of Philosophy

in

Computer Science

Stony Brook University

2015

Binary instrumentation has assumed an important role in software security, as well as related areas such as debugging and monitoring. Binary instrumentation can be performed statically or dynamically. Static binary instrumentation (SBI) is attractive because of its simplicity and efficiency. However, none of the previous SBI systems support secure instrumentation of COTS binaries. This is because of several challenges including: (a) static binary code disassembly errors, (b) difficulty of handling indirect control flow transfers, (c) ensuring completeness of instrumentation, i.e., instrumenting all of the code, including code contained in system libraries and compiler-generated stubs, and (d) maintaining compatibility with complex code, i.e., ensuring that the instrumentation does not break any existing code.

We have developed a new static binary instrumentation approach, and present a software platform called PSI that implements this approach. PSI integrates a coarse grained control flow integrity (CFI) property as the basis of secure, non-bypassable instrumentation. PSI scales to large and complex stripped binaries, including low-level system libraries. It provides a powerful API that simplifies the development of custom instrumentations.

We describe our approach, present several interesting security instrumentations, and analyze the performance of PSI. Our experiments on several real-world applications demonstrate that PSI's runtime overheads are about an order of magnitude smaller than that of the most popular platforms available today, such as

DynamoRIO and Pin. (Both these platforms rely on dynamic instrumentation.) PSI has been tested on over 300 MB of binaries.

In addition to our platform PSI, we describe two novel security applications developed using PSI. First, we present a comprehensive defense against injected code attacks that ensures code integrity at all times, even against very powerful adversaries. Second, we present a defense against code reuse attacks such as return-oriented programming (ROP) that is effective against adversaries possessing a wide range of capabilities.

Dedicated to my wife, my parents and friends…

Contents

List of Figures

Acknowledgements

First and foremost, I would like to thank my advisor R. Sekar, for his guidance in my Ph.D program. I really appreciate his effort on improving my research skills and ability especially critical thinking. He taught me how to reason a problem from scratch with patience. Without this property, I could not have achieved the work that have been done in this dissertation. In addition to that, I would like to thank my advisor for his patience and consistent support on my study. Without that, I may not establish myself as a researcher so quickly in last four years. Finally, I thank him for the guidance on paper writing and idea selection.

Second, I would like to sincerely thank my committee members, Professor Mike Ferdmen, Professor Michalis Polychronakis and Professor Zhiqiang Lin. Their constructive comments and insightful suggestions have greatly helped improving the quality of my dissertation. In addition to that, I really appreciate their suggestions on potential new ideas and research directions which provide a new perspective for me to continue my work.

Secure systems lab is my academic home. I would like to thank all of my lab mates with whom I had an intersection, Wai-kit Sze, Niranjan Hasabnis, Alireza Saberi, Tung Tran, Rui Qiao, Riccardo Pelizzi, Laszlo Szekeres, Daiping Liu and Yarik Markov for their valuable collaborations and assistance on my research.

Finally, I would like to thank my friends and all the fundings including National Science Foundation (NSF) grant CNS-0831298 and CNS-1319137 ,Air Force Office of Scientific Research (AFOSR) grant FA9550-09-1-0539 and Office of Naval Research (ONR) grant N000140710928, for the support of my research work.

Chapter 1

Introduction

Program instrumentation refers to insertion of additional code to an application in order to measure performance, detect or diagnose errors, and collect trace information [24]. Recently, instrumentation has played a central role in security defenses such as exploit detection/prevention, security policy enforcement, application monitoring and malware analysis.

Security instrumentation may be performed either on source or binary code. Source code instrumentation can be more easily and extensively optimized by exploiting higher level language information such as types. However, binary instrumentations are more widely applicable since end users have ready access to binaries. Moreover, security instrumentation should be applied to all code, including all libraries, inline assembly code, and any code inserted by the compiler/linker. Here again, binary based techniques are advantageous.

1.1 Existing binary instrumentation techniques

Binary instrumentation has been the technique of choice for security instrumentation of Commercial Off-The-Shelf (COTS) binaries. Previous works have used binary based instrumentation for sandboxing [70, 84], taint-tracking [103, 116], defense from return-oriented programming (ROP) attacks [57, 63], and other techniques for hardening benign code [76, 111].

Binary instrumentation can either be static or dynamic. Static binary instrumentation (SBI) is performed off-line on binary files, whereas *dynamic* binary instrumentation (DBI) operates on code already loaded into main memory. DBI techniques disassemble and instrument each basic block just before its first execution. They instrument the original code block and put it into a memory pool called code cache.

Static binary instrumentation has many advantages over dynamic instrumentation. Avoiding the use of code cache is one of them. Unlike DBI, SBI instruments binaries ahead of time, and therefore it has lower overhead on program start. Since there is no code generation at runtime, SBI usually has a smaller memory footprint and thus it is lighter-weight. In addition, SBI avoids the insecurity of a writable and executable code cache. All these benefits contribute to more efficient and more secure instrumentation.

1.2 Challenges of static binary instrumentation

While static binary instrumentation has the benefits of simplicity and being light-weight, existing SBI techniques for security instrumentations face several challenges:

- **Disassembly errors.** Binary instrumentation requires all the machine code to be correctly identified. Otherwise, the instrumentation will be incorrect, and an instrumented program may crash or fail in other ways. Correctly identifying all machine code is very difficult. The fundamental reason is that binaries may have data embedded in the middle of code. Linearly scanning all of the binary will incorrectly disassemble such data as code. Disassembling the binary following the control flow will avoid this problem. However, this method may not identify some of code that is reachable only through indirect control transfers. For these reasons, static binary disassembly remains a challenge. We present a static disassembly algorithm to solve the binary disassembly problems for benign COTS software. The design of this algorithm is based on two observations on x86/Linux: (a) it is feasible to disassemble most part of the binaries using linear disassembler and (b) the disassembly errors can be identified using code pointer analysis. Observation (a) indicates that data embedded in code only causes local disassembly errors which do not propagate to the whole binary. Observation (b) describes a common case that if an instruction sequence is located after a chunk of data and never targeted by direct control flows, it is often incorrectly disassembled due to the misidentification of data-code boundary. In this case, the misidentified instructions can be reliably recovered if code pointers can be discovered.

- **Difficulties on program analysis.** Meaningful binary instrumentation usually requires program analysis to recover high level program semantics. This is achieved by static or dynamic analysis on binaries. However, COTS software are shipped with binaries only without additional information such as debugging metadata, type, relocation or (static) symbol table. Missing of these types of information poses challenges on understanding the program. For instance, identifying code and data, recovering code pointers, function stack frames, local variables, types and even function boundaries all become challenges on COTS binaries. To help solving these issues, we use a conservative static analysis to discover code pointers. A straightforward approach to discover code pointers is to identify all functions whose addresses are taken. This is a reasonable option for binaries with additional high level information. Unfortunately, high level information such as code (function) pointers and symbol addresses are not available in COTS binaries. To cope with that, we presents a conservative static code pointer analysis in Section 2.2. In addition, it leverages some available low level information to recover function boundaries.

- **Instrumentation bypassability.** Instrumentation procedures insert code snippets called instrumentation code [1] into original binaries. These snippets should be non-bypassable to ensure correctness of instrumentation. This is not an issue if only direct control transfers exist in binaries, as their target can be checked at instrumentation time. However, indirect control targets are not known until runtime. Such control transfers may "skip" instrumentation code. Moreover, this bypass can lead to the execution of undiscovered and uninstrumented code, say, by jumping into the middle of a multi-byte instruction. SBI is required to instrument indirect control flows and ensure that they never target unintended locations. To safely instrument indirect control flow transfers, we uses runtime address translation, a concept we have borrowed from dynamic binary instrumentation techniques. The basic idea of our instrumentation is to reconstruct the original binary with

[1] Note that in this dissertation, the term *instrumentation code* refers to the code inserted by instrumentation, while *instrumented code* means code that has been instrumented, i.e., the combination of instrumentation code and original code.

instrumentation code inserted, generate a new code segment and put it at the end of the original binary. Since instrumented code is appended after original binary, it will not overwrite any existing data section. Moreover, original code is left unchanged. This is to make sure that data references targeting original code remain valid. Finally, all direct control flow targets are changed to corresponding locations in new code.

- **Instrumentation transparency.** Instrumentation may change the original program state and generate additional memory footprint. These changes can cause programs to crash or behave differently. To avoid this possibility, instrumentation code should not modify any registers, data or code memory used by original applications i.e., full transparency should be maintained.

  As an example, consider the case where code pointers such as saved return addresses on the program stack are used by an application. Since the instrumentation used by PSI causes instruction locations to change, a straightforward implementation would change these saved return addresses on the stack. Unfortunately, programs use this information in several ways: (a) position-independent code (PIC) computes the locations of static variables from return address, (b) C++ exception handler uses return addresses to identify the function (or more specifically, the try-block within the function) to which an exception needs to be dispatched and (c) a program may use the return address (and any other code pointer) to read constant data stored in the midst of code, or more generally, its own code. Changes to saved return address would cause these uses to break, thus leading to application failure. For this reason, the instrumentation presented in this dissertation is designed to provide full transparency. For instance, the instrumentation only generates original return addresses on the stack. In addition, PSI leaves the original code untouched and makes sure that code pointers in the heap, the stack and the global memory are left untouched as well.

- **Dynamic loading.** Instrumentation should be applied to all the running modules of target programs. This is not a problem for statically linked binaries because only one module will be loaded at runtime. However, the vast majority of deployed software is in the form of dynamically linked binaries. Handling dynamically linked binaries requires SBI to overcome two challenges: (a) identifying dependencies, and (b) properly connecting instrumentations across modules. Looking into the first issue, we observe that dynamically linked binaries may load a large number of dependent libraries at load time and even at runtime. Failing to instrument dependent libraries may cause the instrumentation to be incomplete, or worse, lead to inconsistencies that cause runtime errors. A straightforward approach is to identify all dependent libraries statically for a given executable binary. Unfortunately, there are two complications here: (a) some of the dependent libraries are not explicitly recorded in the executable files and (b) dependent libraries may be specified by file name but the full directory path is not given. To handle this problem, we leverage two approaches. The first transforms all libraries in known locations such as `/lib` and `/usr/lib` in advance. For applications that may use libraries whose locations may be difficult to predict, on-demand instrumentation is used. The idea is to monitor library loading and instrument libraries on the fly. Libraries once instrumented do not have to be instrumented again on the next run.

- **Easy-to-use API.** Instrumenting an application using DBI tools is as simple as prefixing its invocation with the name of DBI binary. DBI platforms such as Pin [90], DynamoRIO [51] and Valgrind [102] provide a convenient API that greatly simplifies the development of new instrumentations (also called **client tools**). However, existing static tools do not provide an easy-to-use API comparable to mature DBI tools. This limitation impedes the development of static binary instrumentation. We integrate all the techniques required and provides a generic binary instrumentation platform PSI. Similar to DBI tools, PSI has both a low level and a high level API. Low level API allows users to insert inline assembly code, while high level API allows users to intercept events like system calls and signals.

1.3 Contributions

In this dissertation, we present a generic binary instrumentation platform PSI that addresses the shortcomings of previous SBI techniques. The following is a list of contributions of this dissertation.

- **A disassembly algorithm for COTS binaries.** We present a disassembly algorithm that could reliably discover all data in the middle of code. This algorithm is robust on large and complex binaries without the need of symbols, relocations or debugging information.

- **A conservative static code pointer analysis algorithm.** We present a static code pointer analysis algorithm that could discover code pointers in COTS binaries. Experiments demonstrate that the algorithm works well even on complex and low level binaries.

- **A practical control flow policy for complex binaries.** We present a practical CFI policy that works for complex binaries that contain inline assembly, or pure assembly code. Our policy effectively constrains the targets of indirect control flow transfers.

- **Sound instrumentation with security guarantee.** PSI ensures that its instrumentation cannot be bypassed when applied to COTS binaries. We present a proof of the soundness of instrumentation.

- **A metric to evaluate the strength of control flow policy.** Average indirect target reduction (AIR) is a metric presented in this dissertation to quantify the strength of any control flow policy. We demonstrate that the integrated CFI policy in PSI has almost the same strength as the original CFI implementation [28].

- **An efficient and robust implementation.** We show that PSI supports a robust suite of applications. It provides better performance on real world programs over existing dynamic tools such as DynamoRIO [51] and Pin [90]

- **Security applications to against low level attacks.**we have developed two security applications based on the PSI platform. These applications could effectively defeat modern code reuse attacks and code injection attacks.

1.4 Dissertation organization

The rest of this dissertation is organized as follows: Chapter 2 focuses on the discussion of key techniques of static binary instrumentation in PSI. Chapter 3 presents PSI platform with its API, applications and a systematic evaluation. Chapter 4 and Chapter 5 describes two large applications of PSI. In particular, Chapter 4 focuses on the discussion of defeating stealthy ROP attacks while Chapter 5 proposes a comprehensive solution to defeat modern code injection attacks. Chapter 6 presents the related work and Chapter 7 presents the conclusion and future research direction.

Chapter 2

Core Static Binary Instrumentation Techniques

This chapter presents the core technique for realizing secure and scalable static binary implementation. The content of this chapter is divided into several parts including binary disassembly, static analysis, instrumentation steps, address translation as well as a definition of a CFI policy with a metric to evaluate the strength of CFI.

2.1 Binary disassembly

Binary disassembly is the basis of static binary instrumentation. In the following, we first discuss the existing disassembly techniques as well as their limitations. Then, we present a robust binary disassembly algorithm that has been successfully used on large COTS applications such as Adobe Acrobat, and complex binaries such as glibc.

2.1.1 Existing disassembly techniques

Disassembly techniques used by previous research can be categorized into two types: *linear disassembly* and *recursive disassembly.* Linear disassembly starts by disassembling the first instruction in a given segment. Once an instruction at an address l is disassembled, and is determined to have a length of k bytes, disassembly proceeds to the instruction starting at address $l + k$. This process continues to the end of the segment.

Linear disassembly can be confused by "gaps" in code that consist of data or alignment-related padding. These gaps will be interpreted by linear disassembly as instructions and decoded, resulting erroneous disassembly. With variable-length instruction sets such as those of x86, incorrect disassembly of one instruction can cause misidentification of the start of the next instruction; hence these errors can cascade past the end of gaps.

Recursive disassembly uses a different strategy, one that is similar to a depth-first construction of program's control-flow graph (CFG). It starts with a set of code entry points specified in the binary. For an executable, there may be just one such entry point specified, but for shared libraries, the beginning of each exported functions is specified as well. The technique starts by disassembling the instruction at an entry point. Subsequent instructions are disassembled in a manner similar to linear disassembly. The difference with linear disassembly occurs when control-flow transfer instructions are encountered. Specifically, (a) each target identified by a direct control-flow transfer instruction is added to the list of entry points, and (b) disassembly stops at unconditional control-flow transfers.

Unlike linear disassembly, recursive disassembly does not get confused by gaps in code, and hence does not produce incorrect disassembly.[1] However, it fails to disassemble code that is reachable only via indirect control flow transfers (ICF transfers).

Incompleteness of recursive disassembly can be mitigated with a list of all targets that are reachable only via ICF transfers. This list can be computed from *relocation information*. However, in stripped binaries, which typically do not contain relocation information [2], recursive disassembly can fail to disassemble significant parts of the code.

[1]This does rely on some assumptions: (a) calls must return to the instruction following the call, (b) all conditional branches are followed by valid code, and (c) all targets of (conditional as well as unconditional) direct control-flow transfers represent legitimate code. These assumptions are seldom violated, except in case of obfuscated code.

[2]Relocation information is used for linker to link different object files into an executable and it is no longer needed when an executable has been generated

2.1.2 PSI disassembly algorithm

The above discussion on using relocation information to complete recursive disassembly suggests the following strategy for disassembly:

- Develop a static analysis to compute ICF targets.
- Modify recursive disassembly to make use of these as possible entry points.

Unfortunately, the first step will typically result in a superset of ICF targets: some of these locations do not represent code addresses. Thus, blindly following ICF targets computed by static analysis can lead to incorrect disassembly. We therefore uses a different strategy, one that combines linear and recursive disassembly techniques, and uses static analysis results as positive (but not definitive) evidence about correctness of disassembly. PSI starts by eagerly disassembling the entire binary using linear disassembly, which is then checked for errors. The error checking step primarily relies on the steps used in recursive disassembly. Finally, an error correction step identifies and marks regions of disassembled code as representing gaps. The error detection step relies on the following checks:

- *Invalid opcode:* Some byte patterns do not correspond to any instruction, so attempts to decode them will result in errors. This is relatively rare because x86 machine code is very dense. But when it occurs, it is a definitive indicator of a disassembly error.
- *Direct control transfers outside the current module.* Cross-module transfers need to use special structures called program-linkage table (PLT) and global offset table (GOT), and moreover, they need to use ICF transfers. Thus, any direct control transfer to an address outside the current module indicates erroneous disassembly.
- *Direct control transfer to the middle of an instruction:* This can happen either because of incorrect disassembly of the target, or incorrect disassembly of the control-flow transfer instruction. Detection of additional errors near the source or target will increase the confidence regarding which of the two has been incorrectly disassembled. In the absence of additional information, PSI approach considers both possibilities.

Since errors in linear disassembly arise due to gaps, the error correction step relies on identifying and marking these gaps. An incorrectly disassembled instruction signifies the presence of a gap, and its beginning and end need to be found. To find the beginning of the gap, PSI simply walks backward from the erroneously disassembled instruction to the closest preceding unconditional control-flow transfer. If there are additional errors within a few bytes preceding the gap, the scan is continued for the next preceding unconditional control-flow transfer. To find the end of the gap, PSI relies on static analysis results (Section 2.2). Specifically, the smallest ICF target larger than the address of the erroneously disassembled instruction is assumed to be the end of the gap. Once again, if there are disassembly errors in the next few bytes, PSI extends the gap to the next larger ICF target.

After the error correction step, all identified disassembly errors are contained within gaps. At this point, the binary is disassembled again, this time avoiding the disassembly of the marked gaps. If no errors are detected this time, then the work is done. Otherwise, the whole process needs to be repeated. While it may seem that repetition of disassembly is an unnecessarily inefficient step, PSI has used it because of its simplicity, and because disassembly errors have been infrequent enough in the implementation that no repetitions are needed for the vast majority of the benchmarks.

2.1.3 Disassembler implementation

Binaries on Linux (and most other UNIX systems) use the ELF (Executable and Linkable Format) [137] format. PSI supports binaries that represent executables and shared libraries. The ELF format divides a binary into several sections, each of which may contain code, read-only data, initialized data, and so on. While PSI utilizes the data in read-only data sections, it is mainly concerned with the code sections.

A typical executable contains the following code sections: `.init`, `.plt`, `.text` and `.fini`. However, shared libraries and atypical executables may have a different set of code sections. Instead of making assumptions about the names of code segments, this approach obtains the list of all executable segments from the ELF header, and proceeds to disassemble and instrument each of them.

PSI implementation utilizes `objdump` to perform linear disassembly. PSI has built a disassembly error detection and correction components on top of `objdump`. In the experiments, disassembly errors occurred primarily due to insertion of null padding generated by legacy code or linker script.

In addition, PSI discovers jump table data in the middle of code in `libffi.so` and `libxul.so`

There were also several instances where conditional jumps targeted the middle of an instruction. Further analysis revealed that these errors occurred with instructions that had optional prefixes, such as the "lock" prefix. PSI eliminates this error by treating these prefixes as independent instructions, so that jumps could target the instruction with or without the prefix.

2.2 Binary analysis

Static binary analysis is an essential technique for binary instrumentation. Binary analysis could be used for various purposes, including disassembly of binary code and discovery of high level information such as function boundary, code pointers as well as local variables and type information.

Static binary analysis on code pointers is a critical technique since it provides (a) the important information that helps discover code reachable only though indirect control transfers and (b) an abstract control flow graph that helps analyzers to better understand program control flow.

This section proposes a static analysis method for discovering possible indirect control flow (ICF) targets without additional information. The key observation is that *almost all code pointers take the form of a constant whose value points to a valid instruction boundary.* This approach classifies ICF targets into several categories, and devises distinct analyses to compute them:

- *Code pointer constants (CK)* consist of code addresses that are constants at compile-time.
- *Computed code addresses (CC)* include code addresses that are computed at runtime.

- *Exception handling addresses (EH)* include code addresses that are used by exception handlers.
- *Exported symbol addresses (ES)* include export function addresses.
- *Return addresses (RA)* include the code addresses next of a call.

Static analysis results are filtered to retain only those addresses that represent valid instruction boundaries in disassembled code. Observation on code pointers remains true except for the computed code addresses (CC), which will be further discussed in Section 2.2.2.

2.2.1 Identifying code pointer constants

In general, there is no way to distinguish a code pointer from other types of constants in code. So, PSI takes a conservative approach. Any constant that "looks like a code pointer," as per by the following tests, is included in CK.

- it falls within the range of code addresses in the current module, and
- it points to an instruction boundary in disassembled code.

Note that a module has no compile-time knowledge of addresses in another module, and hence it suffices to check for constants that fall within the range of code addresses in the current module. For shared libraries, absolute addresses are unknown, so PSI checks if the constant represents a valid offset from the base of the code segment. It is also possible that the offset may be with respect to the GOT of the shared library, so the validity check takes that into account as well.

The entire code and data segments are scanned for possible code constants as determined by the procedure in the preceding paragraph. Since 32-bit values need not be aligned on 4-byte boundaries on x86, PSI uses a 4-byte sliding window over the code and data to identify all potential code pointer constants.

2.2.2 Identifying computed code pointers

Whereas the CK analysis was very conservative, it is difficult to bring the same level of conservativeness to the analysis of computed code pointers. This is because, in general, arbitrary computations may be performed on a constant before it is used as an address, and it would be impossible to estimate the results of such operations with accuracy. However, these general cases are just a theoretical possibility. The vast majority of code is generated from high-level languages where arbitrary pointer arithmetic on code pointers isn't meaningful.[3] Even for hand-written assembly, considerations such as maintainability, reliability and portability lead programmers to avoid arbitrary arithmetic on code pointers. So, rather than supporting arbitrary code pointer computation, PSI supports computed code pointers in a limited set of contexts where they seem to arise in practice. Indeed, the only context in which the author has observed is that of jump tables.[4]

The most common case of jump tables arises from compiling switch statements in C and C++ programs. If these were the only sources of CC, then a simple approach could be developed that is based on typical conventions used by compilers for translating switch statements. However, jump tables in hand-written assembly take diverse forms. So, we begin identifying properties that are likely to hold for most jump tables:

- Jump table targets are intra-procedural: the ICF transfer instruction and ICF target are in the same function. (function boundaries are not required — they are estimated conservatively, as described below.)
- The target address is computed using simple arithmetic operations such as additions and multiplication.
- Other than one quantity that serves as an index, all other quantities involved in the computation are constants in the code or data segment.
- All of the computation takes place within a fixed size window of instructions, currently set to 50 instructions in PSI implementation.

Based on these characteristics, PSI uses a static analysis technique to compute possible CC targets. It uses a three-step process. The first step is the identification of function boundaries and the construction of a control-flow graph. In the absence of full symbol table information, it is difficult to identify all function boundaries, so it falls back to the following approach that uses information about exported function symbols. PSI treats the region between two successive exported function

[3]This is true even in languages that are notorious for pointer arithmetic, such as C.

[4]C++ exception handling also involved address arithmetic on return addresses, but PSI can rely on exception handler information that must be included in binaries rather than the CC analysis described in this section

symbols as an approximation of a function. (Note that this approximation is conservative, as there may be non-exported functions in between.) Then a control-flow graph is constructed for each region.

In the second step, PSI identifies instructions that perform an indirect jump. It performs a backward walk from these instructions using the CFG. All backward paths are followed, and for each path, PSI traces the chain of data dependences to compute an expression for the indirect jump target. This expression has the form $*(CE_1 + Ind) + CE_2$, where CE_1 and CE_2 denote expressions consisting of only constants, Ind represents the index variable, and $*$ denotes memory dereferencing. In some cases, it is possible to extend the static analysis to identify the range of values that can be taken by Ind. However, PSI has not implemented such an analysis, especially because the index value may come from other functions. Instead, this work makes an assumption that valid Ind values will start around 0.

In the third step, PSI enumerates possible values for the index variable, computes the jump target for each value, and check if it falls within the current region. Specifically, PSI checks if $CE_1 + Ind$ falls within the data or code segment of the current module, and if so, retrieve the value stored at this location. It is then added with CE_2 and the result checked to determine if it falls within the current region. If so, the target is added to the set CC. If either of these checks fail, Ind value is deemed invalid.

The approach starts from Ind value of 1, and explores values on either side until finding a value for which the computed target is invalid is reached.

Note that the backward walk through the CFG can cross function boundaries, e.g., traversing into the body of a called function. It may also go backwards through indirect jumps. To support this case, PSI extends the CFG to capture indirect jumps discovered by the analysis. The maximum extent of backward pass is bounded by the window size specified above.

The above procedure may potentially fail in extreme cases, e.g., if CC computation is dispersed beyond the 50-instruction window used in the analysis, or if the computation does not have the form $*(CE_1 + Ind) + CE_2$. In such cases, PSI can conservatively add every instruction address within the region to CC. In the experiments, no such exception has been found.

2.2.3 Identifying other code addresses

Below, we describe the computation of the three remaining types of code pointers: exception handlers (EH), exported symbols (ES), and return addresses (RA).

In ELF binaries, exception handlers are also valid ICF targets. They are constructed by adding a base address with an offset. The base addresses and offsets are stored in ELF sections `.eh_frame` and `.gcc_except_table` respectively. Both these sections are in DWARF [129] format. PSI uses an existing tool, katana [107, 118], to parse these DWARF sections and get both base addresses and offsets, and thus compute the set EH. (Note that the CC analysis mentioned above won't be able to discover these EH targets because DWARF format permits variable length numeric encoding such as `LEB128`, and hence the simple technique of scanning for 32-bit constant values won't work.)

Exported symbol (ES) addresses are listed in the dynamic symbol table, which is found in the `.dynamic` section of an ELF file.

Return addresses (RA) are simply the set of locations that follow a call instruction in the binary. Thus, they can be computed following the disassembly step.

2.3 Binary instrumentation

After disassembly and code pointer analysis, the resulting code is ready to be instrumented. This section presents the instrumentation steps used by PSI.

2.3.1 Background

Dynamic Binary Instrumentation Binary instrumentation can be static and dynamic. Dynamic methods instrument binary code at runtime while the static ones do it before runtime. DBI tool works as follows: it runs as an interpreter by translating source binary code into new code of target architecture. Instrumentation is performed in the meantime before the new code is generated. Generated code is in the granularity of basic blocks which are allocated into a memory region called code cache used for program execution. DBI executes an application by executing basic blocks in code cache instead of native application code.

Basic blocks are connected with direct branches to avoid frequent context switch between DBI runtime and target application execution. To ensure that DBI never loses control of program execution, indirect branches must be translated to make sure their targets point to code cache instead of code in original program. This is achieved by replacing each indirect branch into a sequence of instructions that performs a hash lookup. The lookup checks against a hash table where the key is original address pointing to application code, and the value is the corresponding address pointing to code cache. Successful lookup diverts control to target code cache block and program continues, while lookup failures redirects control to DBI runtime, where a new basic block will be generated and its entry address added into the hash table before program execution proceeds.

Static Binary Instrumentation Static binary instrumentation inserts instrumentation code statically into target binary. This can be realized in several approaches. One intuitive option is to insert instrumentation code in place between original code snippets. This approach is used in link time instrumentation efforts such as PLTO [121] and Abadi CFI [28]. However, symbols and relocations are essentially needed to relocate original code. Since COTS binaries do not have such information, this approach becomes less applicable. Another option is to patch original instructions with direct jump and put instrumentation code in other locations. Previous work such as detour [77], CCFIR [152], LEEL [145], binaryRAD [115] adopts this approach. However, this approach may cause issues because patching instructions may overwrite succeeding instructions that are also control transfer targets [50]

Similar to DBI, static binary instrumentation can be realized without patching original code. One of the examples is REINS [141] which targets sandboxing of untrusted COTS executables on Windows. Different from static tools that patch original code, it uses out-of-line instrumentation where instrumented code is generated and appended after original binary, original code is preserved at runtime. All indirect control targets are changed to valid locations in instrumented code. Since original code is maintained, any data references to original code and data are preserved to ensure correctness of execution. By doing so, REINS ensures that sandboxed code can never escape instrumentation (except to invoke certain trusted functions). However, ability of instrumenting dependent libraries used by the application is not clearly mentioned as a supported feature. Moreover, their evaluation does not consider as many large applications.

2.3.2 Binary instrumentation in PSI

PSI is a static instrumentation tool but uses out-of-line instrumentation similar to DBI tools. In addition, PSI supports all shared libraries and works with complex COTS binaries. The instrumentation steps include: 1) instrumentation, 2) reassembling and 3) binary re-generation.

Instrumentation is performed on assembly representation. This simplifies the implementation since it does not need to be concerned with details such as encoding and decoding of instructions. Moreover, it can use labels instead of addresses. In particular, for each instruction location A in the disassembler output, PSI associates a symbolic label `L_`A as follows:

`L_8040930:`movl %ecx, %eax

These symbolic labels are used as targets of direct branch instructions, which means that the assembler will take care of fixing up the branch offsets. (These offsets will typically change since PSI is inserting additional code during instrumentation.)

After rewriting, the instrumented assembly file is processed using the system assembler (the GNU assembler *gas*) to produce an object file. PSI then extracts the code from this object file and then uses the *objcopy* tool to inject it into the original ELF file. Note that the original code sections are preserved. This ensures that any attempt by the instrumented program to read its own code will produce the same results as the original program.

In the final step, we prepare the ELF file produced by objcopy for execution. This step requires relocation actions on the newly added segment, and updating the ELF header to set its entry point to the segment containing instrumented code. All original code segments are made non-executable. For shared libraries, it is also necessary to update the dynamic symbol table section.

2.3.3 Address translation for indirect control transfer

As described above, instrumented code resides in a different code segment (and hence a different memory location) from the original code. This means that function pointer values, which will typically appear in the code as constants, will have incorrect values. Unfortunately, it is not possible to fix them up automatically, since PSI cannot distinguish constants representing code addresses from other types of constants. It would obviously be unsound to modify a constant value that does not represent a code pointer.[5]

The typical way to deal with this uncertainty, employed in DBI [51], is to wait until a value is used as the target of an ICF transfer. At that point, this target value is translated into the corresponding location in the instrumented code. This translation is performed using a table that consists of pairs of the form

$$\langle \text{original address, new address} \rangle$$

At runtime, `addr_trans`, a piece of trampoline code, performs address translation.

```
                                  L_060c0: push $060c2
                                           movl %eax, %gs:0x44
    060c0:   call *%ecx                    movl %ecx, %eax
    060c2:   ......                        jmp addr_trans
                                  L_060c2: ......
```

FIGURE 2.1: Original (left) and Instrumented code (right) for ICF transfer

This code saves the register (%eax) used by the instrumentation, and moves the target address into it.[6] Then the original indirect jump (or call) is replaced with a direct jump to the trampoline routine, `addr_trans`. Note the use of labels such as `L_060c0` that are used to associate locations in the instrumented code with the corresponding original address, namely, `060c0`. As a result, the translation table can consist of entries of the form

$$\langle A, \texttt{L\_}A \rangle$$

for each valid ICF target A. The details of `addr_trans` are as follows: After saving registers and flags needed for its operation, `addr_trans` performs an address range check to determine if the target is within the current module. If not, this represents a cross-module control transfer that is discussed later in this section. After the range check, `addr_trans` performs address translation. PSI implementation relies on open hashing [142] to perform an efficient lookup of the table described above.

[5]Here again, relocation information can address this uncertainty, but in the case of PSI, this is unavailable.

[6]Note that %gs points to the base of thread-local storage, and %gs:0x44 is not used by existing system software.

Rather than storing just the target address `L_`A in the table, PSI stores code that transfers control to `L_`A. For instance, the hash table entry to translate a code address `0x060c2` looks as follows.

`0x060c2`	`movl %gs:0x44, %eax; jmp L_060c2`

If no translation is found for the target address, `addr_trans` will set an error code to help in debugging, and terminate the program.

Note that, for shared libraries, the translation table contains only the offsets but not absolute addresses. Consequently, the base address of the module needs to be subtracted from the runtime address given to the translation routine. PSI relies on the dynamic linker to patch the routine with the module's base address when the module is loaded.

In order to preserve the functionality of original code, it is necessary to ensure that the instrumentation does not modify any of the registers or memory used by the program. It is relatively easy to avoid changes to memory, or registers other than the program counter (PC). Since instrumentation changes code locations (as described earlier), it is not possible to preserve the PC register. So, what PSI needs to do is to add a compensation for any operation that uses the PC for any purpose other than fetching the next instruction. Fortunately, on 32bit x86, there are only two instructions that use PC this way: call and return. A `callX` is translated into a `pushnext`; `jmpX`, where `next` denotes the address of the instruction following `call` in the original program. Similarly, a return is translated into a `pop` followed by a direct jump. Note that after this transformation, none of the instructions in the original program involve movement of data between PC and other registers or memory, thus ensuring that program behavior is unaffected by PSI instrumentation. In x86-64 architecture, any PC-relative data addressing [7] needs to be translated too. This can be done by either modifying the offset value or using a dedicated register. Section 4.5.2.1 conveys some of the details on how to port the instrumentation to x86-64.

2.3.4 Signal handling

Signals are another mechanism to redirect program control flow. If a program registers its signal handlers, once again PSI will have the problem that the program

[7]instructions using RIP register

will specify the location of the handler in original code, whereas the platform wants the signal to be delivered to the instrumented code. (This problem arises because signals are delivered by the kernel, which is unaware of the address translations used to correctly handle code pointers.)

PSI implementation intercepts `sigaction` and `signal` system calls, and stores the address of the signal handlers specified by these calls in a table. The signal handler argument is then changed so that control will be transferred to a wrapper function, which contains code that jumps to the user-specified handler. Since this wrapper will be instrumented as usual, instrumented version of the user-specified handler will be invoked.

2.3.5 Modular instrumentation

Software is typically organized into dynamically linked binaries. This means dependent library code address may or may not be known before runtime. This calls for independent instrumentation for each module as well as handling dynamic libraries whose base address may be randomized. This section focuses on the techniques required for modular instrumentation

Shared Library Support for shared libraries is achieved as follows. PSI rewrites a single module (an executable or a shared library) at a time. There is exactly one version of a transformed shared library, regardless of the context (or the executable) in which it is used. Note that PSI transforms all shared libraries, including `glibc` and `ld.so`.

As described before, `addr_trans` already handles intra-module control transfers. Inter-module transfers rely on a two-stage process. In the first stage, a *global translation table (GTT)* is used to map an ICF target to the translation routine address in the target module. This table is constructed as follows. Since shared libraries must begin at page boundaries, any two modules have to be apart by at least 4KB, the page size on 32-bit Linux systems. Thus, it is enough to use the leading 20 bits of the ICF target in this lookup table. PSI uses a simple array implementation for GTT since there are only $2^{20} = 1M$ entries in this table. This array is made read-only in order to protect it. The second stage performs a lookup in the destination module, using the address translation table for that module. In this dissertation, we use the term *module translation table (MTT)* for the translation table that specifies translations for addresses within the module.

Changes to the Loader Note that the GTT needs to be updated as and when modules are loaded. Naturally, the best place to do this is the dynamic linker. This is accomplished by modifying the source code of `ld.so`. The change uniformly handles the typical case of the loader mapping all of shared libraries referenced by an executable (or another shared library loaded by the loader), as well as the less common case of an application using `dlopen` and `dlclose` primitives to load and unload libraries at runtime. PSI changes relate to about 300 lines of the source code of `ld.so`.

PSI loader modification also addresses two other idiosyncrasies of `ld.so`. First, note that this approach modifies the entry point of a binary. Thus, any program that uses the entry point for purposes other than jumping to it may not work any more. As it turns out, `ld.so` does make use of this information when it is invoked to load a program, as in `ld.so <binary>`. The loader is changed so that it compensates for this change in the entry point, and hence works correctly in all cases. We defer the description of the 2nd idiosyncrasies into Section 2.5.3

2.4 Optimizations

Runtime performance is a critical metric to evaluate a binary instrumentation system. In PSI, runtime overhead mainly comes from I-cache pressure, D-cache pressure and branch prediction errors. The source of I-cache pressure comes from the additional instructions executed for handling indirect branches, while D-cache pressure is due to the additional data references to address translation table. Branch prediction error happens mostly because branch predictors for indirect branches are simply useless, since all indirect branches of one module in PSI share one code piece.

We discuss the optimizations performed to address these performance bottlenecks. In particular, Section 2.4.1 demonstrates how to improve branch prediction despite the translation. Section 2.4.2 demonstrates how to reduce I-cache and D-cache pressure by avoiding address translation. Section 2.4.3 shows how to reduce I-cache pressure by eliminating some instrumentation code for transparency.

2.4.1 Improving branch prediction

Modern processors use very deep pipelines, so branch prediction misses can greatly decrease performance. Unfortunately, translation of returns (into a combination of pop and jmp) in PSI leads to misses. When a return instruction is used, the processor is able to predict the target by maintaining a stack (Return Stack Buffer) that keeps track of calls. When it is replaced by an indirect jump, especially one that is always made from a single trampoline routine, branch prediction for return is not used..

To address this problem, PSI modifies the transformation of calls and returns as shown in Figures 2.2 and 2.3. The original call is transformed into another call into stub code that is part of the instrumentation. There is a unique stub for each call site. The code in the stub adjusts the return address on the stack so that it will have the same value as in the untransformed program. This requires addition of a constant that represents the offset between the call instructions in the original and transformed code. At the time of return, the return address on the stack is translated from its original value to the corresponding value in the transformed program, after which a normal return can be executed.

The key point about this transformation is that the processor sees a return in Figure 2.3 that returns from the call it executed (Figure 2.2, label `L_060b1`). Although the address on the program stack was adjusted (Figure 2.2, label `S_060b1`), this is reversed by address translation in Figure 2.3. As a result, the processor's predicted return matches the actual return address on the stack.

The side effect of this optimization is the introduction of an extra code stub for each call instruction, which increases pressure on I-cache and introduce an additional direct jump. Despite that, we will show in Section 3.5.3 that the benefits of correct branch prediction on return instructions outweighs the side effect.

```
                          L_060b1: call S_060b1
060b1: call 060c0              .....
     .....                S_060b1: add $offset, (%esp)
                                   jmp L_060c0
```

FIGURE 2.2: Optimized instrumentation of calls

`060d1:  ret`	`....` #address translation
	`add $4, %esp`
	`mov %edx, (%esp)`
	`ret`

FIGURE 2.3: Optimized instrumentation of returns

2.4.2 Avoiding address translation

In this dissertation, we explored three optimizations aimed at eliminating address translation (AT) overheads in the following cases:

AT.1 jump tables

AT.2 PIC translation

AT.3 return target speculation

For the first optimization, instead of computing an original code address and then translating it into new addresses, PSI creates a new table that contains translated addresses. The content of the table is copied from the original table, and then each value is translated (at instrumentation time) into the corresponding new address. A catch here is that the size of the original table is unknown. Note, however, that a good guess can be made, based on the CC computation technique from Section 2.2.2. PSI first checks that the index variable is within this range, and if so, use the new table. Otherwise, the work uses the old table, and translate the jump address at runtime.

PIC has several code patterns, including a call to `get_pc_thunk` and a call to the next instruction. The basic function of the pattern is getting the current PC and copying it into a general purpose register. In the translated code, however, `get_pc_thunk` introduces an address lookup for return. This extra translation could be avoided by translating this version into a call of the next instruction. No returns are used in this case, thereby avoiding address translation overhead. (It is worth noting that using a call/pop combination does not affect branch prediction for return instructions. The processor is able to correct for minor violations of call/return discipline.

In the third case, if a particular ICF transfer tends to target the same location most of the time, PSI can speed it up by avoiding address translation for this location. Instead, a comparison is introduced to determine if the target is this location, and if so, introducing a direct jump. In the implementation, we choose to apply it only to return instruction. PSI used profiling to determine if the return frequently targets the same location.

2.4.3 Removing transparency

Using static analysis results, PSI can safely avoid some of the overheads associated with full transparency. We call this removal of transparency as RT. The following are two optimizations PSI uses:

RT.1 no saving of eflags

RT.2 use non-transparent calls

To achieve, RT.1, we analyze all potential indirect and direct control targets. If there is no instruction that uses `eflags` prior to all instructions that define it, then RT.1 can be safely used. In fact, we discover that `eflags` is live only in a few jump tables.

When RT.2 is enabled, all return addresses are within the new code. Note that RT.2 is always enabled on PIC patterns, i.e., call of `get_pc_thunk` and call of next instruction. This is because it is simple to analyze this pattern and determine that the non-transparent mode will not lead to any problems, as long as the offset added to obtain data address is appropriately adjusted.

2.5 Control flow integrity policy

So far, we have discussed all the underlying techniques including binary disassembly, static analysis, instrumentation, address translation as well as optimizations. With this set of static binary techniques, PSI safely handles large and complex binaries and makes sure indirect control flows are correctly handled. In this section, we will focus on the discussion of Control flow integrity (CFI).

2.5.1 Definition of control flow integrity

Control flow integrity (CFI) is a very important low level property that can be achieved using binary instrumentation. CFI has two level of meanings as following.

The first meaning of Control flow integrity is that CFI is a program state in which all program control flows follow the semantics of application. The CFI security policy dictates that software execution must follow a path of a control-flow graph(CFG) determined ahead of time [28]. The CFG could be generated from source code analysis and binary analysis. The second meaning of CFI is that CFI is a low level instrumenation that instruments all indirect control flow transfers and ensures that they all target locations specified by the CFI policy.

CFI policies serve two purposes on the high level: (a) securing target programs from exploits and attacks and (b) ensuring that all low level instrumentations built on top are non-bypassable. In the following subsections, we will discuss them separately.

2.5.2 CFI policies and strength evaluation

One of the most important goals of PSI is to provide security for target applications. However, using address translation of indirect control flows is still too permissive in the sense that the policy allows that any indirect branch could target all code addresses inside a binary. Tightening the policy definitely can help reduce the space of attacks, but over constraining control flow may cause programs to crash due to various corner cases in COTS binaries.

This calls for an effective control flow integrity (CFI) policy which leads to the following two questions: *how to design a CFI policy to cope with complex and large COTS binaries? how to quantify the strength of CFI properties enforced?*.

In the following subsections, we present different types of CFI policy designs with discussion of their applicability to large COTS binaries. Then, we propose a metric to evaluate the strength of each design.

Reloc-CFI: CFI was proposed for the first time by Abadi et. al [28]. The implementation of their CFI policy as well as some of others [47**?**] are generally based on the following model of how ICF transfers are used in source code:

1. *Indirect call (IC):* An indirect call can go to any function whose address is taken, including those addresses that are implicitly taken and stored in tables, such as virtual function tables in C++.

2. *Indirect jump (IJ):* Since compiler optimizations[8] can replace an indirect call (IC) with an indirect jump (IJ), the same policy is often applied to indirect jumps as well.

3. *Return (RET):* Returns should go back to any return address (RA), i.e., an instruction following a call.

It is theoretically possible to further constrain each of these sets, and moreover, use different sets for each ICF transfer. However, implementations typically do not use this option, as increased precision comes with certain drawbacks. For instance, the callers of functions in shared libraries (or dynamically linked libraries in the case of Microsoft Windows) are not known before runtime, and hence it is difficult to constrain their returns more narrowly than described above. Moreover, some techniques rely on relocation information, which does not distinguish between targets reachable by IC from those reachable by indirect jumps, or between the targets reachable by any two ICs. Hence they do not refine over the above property. For this reason, we refer to the above CFI property as reloc-CFI.

The description of implementation in Abadi et al [28] indicates their use of relocation information, and confirms the above policy regarding ICs. No specifics are provided regarding IJs and returns. Note that Li, Wang et. al. [89] use a single table for ICs and IJs, and another for returns, enforcing reloc-CFI but in a kernel environment.

One of the critical limitations of Reloc-CFI is that it requires relocation information which is not available in COTS binaries.

Strict-CFI: Strict-CFI is derived from reloc-CFI, except that it uses ICF targets computed by the ICF target analysis in Section 2.2 rather than relocation information. In addition, strict-CFI incorporates an extension needed to handle features such as exception handling and multi-threading. Specifically, these features are used by a handful of instructions in system libraries, and we simply relax the above policy for these instructions:

[8]Specifically, a tail call optimization that replaces a call occurring at the very end of a function with a jump.

- Instructions performing exception related stack unwinding are permitted to go to any exception handler landing pad (EH).
- Instructions performing context switches are permitted to use any type of ICF transfer to transfer to a function address.

Strict-CFI solves the problems of missing relocation in COTS binaries by using static analysis. However the limitation of strict-CFI is that it handles exception cases specifically for special system libraries. Those exception cases may be common in other COTS binaries where strict-CFI may not handle them in general.

2.5.3 Coarser grained CFIs

Corner cases of control flow transfer in COTS binaries can be handled in general by using coarser grained CFI policies. For instance, the most basic CFI enforcement policy is to allow ICF transfers to touch any instruction in the program. It is regarded as instr-CFI. Obviously, instr-CFI is weak in front of attacks. However, one important property of instr-CFI is ICF transfers never go to the middle of an instruction and cannot bypass any checking code.

Native Client [146] and PittSFIeldCISC-SFI use instruction bundling to constrain targets of ICF transfers. This type of CFI is regarded as bundle-CFI.

BinCFI: Our Proposal Complex binaries can contain exceptions to the simple model of ICF transfers outlined earlier. To define a suitable CFI property for such binaries, in this dissertation, we introduce a category of ICF transfer in addition to RET, IC and IJ described earlier. This category, called PLT, includes all ICF transfers in the program linkage table, a section of code used in dynamic linking.[9] We define BinCFI as shown in Figure 2.4.

It is easy to see that strict-CFI is stricter than BinCFI. The reasons for relaxing strict-CFI are as follows. In general, there is no easy way to distinguish between returns used for purposes such as stack unwinding, longjmp, thread context switch, and function dispatch from (the more common) use of returning from functions. Therefore returns are permitted to go to any of the valid targets corresponding to each of these uses. Returns are some times broken up into a pop and jump,

[9]Specifically, for each function belonging to another module, a stub routine is created by the compiler in this section.

	Returns (RET), Indirect Jumps (IJ)	PLT targets, Indirect Calls (IC)
Return addresses (RA)	Allow	
Exception handling addresses (EH)	Allow	
Exported symbol addresses (ES)		Allow
Code pointer constants (CK)	Allow	Allow
Computed code addresses (CC)	Allow	Allow

FIGURE 2.4: BinCFI Property Definition

so all possible targets of RET are permissible targets of IJ. This explains the first column of the table.

Since the purpose of PLT stubs is to dispatch cross-module calls, it would seem that the targets can only be exported symbols from other modules. However, recent versions of gcc support a new function type called `gnu_indirect_function`, which allows a function to have many different implementations, with the most suitable one selected at runtime based on factors such as the CPU type. Currently, many glibc low level functions such as `memcpy`, `strcmp` and `strlen` use this feature. To support this feature, a library exports a chooser function that selects at runtime which of the many implementations is going to be used. These implementation functions may not be exported at all. To avoid breaking such programs, the policy for PLT should be relaxed to include code pointers in the target library. This is what is done on the second column of Figure 2.4.

Indirect calls should go to the targets in one of the sets CC or CK. Since these two sets are usually much larger than ES, the policy design chose to merge IC and PLT to use the same table of valid targets.

This relaxed form of CFI policy covers almost all the corner cases except one. The experiments discover that the use of return instructions for lazy symbol resolving in dynamic loader violates this policy. Lazy symbol binding is performed by the `_dl_runtime_resolve` function (or `_dl_runtime_profile` if profiling is enabled) in `ld.so`. This function computes the target address corresponding to the symbol, pushes this address on the top of stack, and returns. For this to work correctly, returns should be permitted to target exported symbols, further decreasing the accuracy of the CFI implementation. Instead, the loader is modified to use indirect jumps instead of returns, and the target of these jumps is restricted with the policy shown in Figure 2.4 in Page 28 for PLT entries.

AIR: A Metric for Measuring CFI Strength Previous works on CFI have relied on analysis of higher level code to effectively narrow down ICF targets. Since binary analysis is generally weaker than analyses on higher-level code, the CFI enforcement in PSI is likely to be less precise. It is natural to ask how much protection is lost as a result. To answer this question, we define a simple metric for representing the quality of protection offered by a CFI technique.

Definition 2.1 (Average Indirect target Reduction (AIR)). Let $i_1, ..., i_n$ be all the ICF transfers in a program and S be the number of possible ICF targets in an unprotected program. Suppose that a CFI technique limits possible targets of ICF transfer i_j to the set T_j. In this dissertation, we define AIR of this technique as the quantity

$$\frac{1}{n}\sum_{j=1}^{n}\left(1 - \frac{|T_j|}{S}\right)$$

where the notation $|T|$ denotes the size of set T.

On x86, where branches can target any byte offset, S is the same as the size of code in a binary. In this dissertation, we will measure the strength of CFI in Section 3.5.2.1.

2.5.4 Non-bypassable instrumentation

In previous subsections, we presented the definition of CFI as well as a practical CFI policy for COTS binaries. In this subsection, we focus on the other purpose of CFI, which is instrumentation safety. In fact, CFI implies the property that its instrumentation code added is non-bypassable. This is enforced by the following policies:

- All direct and indirect control-flow transfers made from the original code must target instructions in the original code that were validly disassembled by the disassembler.
- If a snippet was specified for insertion before an instruction I, then all (direct or indirect) control-flow transfers targeting I will instead be made to target the first instruction of the added snippet.

- Only the added instrumentation code can transfer control to libraries containing instrumentation support functions. (Recall that the high-level instrumentation API relies on inserting calls to this library.)

All indirect branches, including jumps, calls and returns, are checked at runtime to ensure the above properties. Direct branches are checked at the time of generating the instrumented binary. A modified loader (Section 2.3.5) denies requests for loading uninstrumented libraries. This ensures that all indirect branches are checked.

The above properties are ensured by a coarse-granularity CFI policy. Built on top of that, PSI has leveraged such a CFI property with additional restrictions intended to ensure the safety of added instrumentation. The following is a non-exhaustive list of attacks that“escape” instrumentation prevented by PSI:

- *Branching to data segments.* As described above, the list of valid targets can only include addresses within validly instrumented code. Thus, data segments cannot appear in this table of valid targets.
- *Branching to code sections that were not recognized and instrumented.* If the disassembler fails to recognize some code fragments, they won't be instrumented. However, since branch targets are restricted to be valid instruction boundaries in disassembled code, any attempt to execute undiscovered code will be blocked.
- *Branching to middle of instructions.* Code reuse attacks are a prime example here. Since the targets are checked to be valid instruction boundaries, these attacks are stopped.
- *Bypassing the instrumentation code.* As noted above, if an instrumentation snippet was specified for insertion before an instruction, that instruction is no longer permitted to be a branch target.
- *Corrupting the integrity of instrumentation logic by jumping into its middle, or by accessing functions intended to be used exclusively by instrumentation.* As noted above, branches will be checked to preclude these targets.

Note that checks on indirect branches protect explicit control flows. implicit control flows are protected by our signal handlers, mentioned in Section 2.3.4

2.5.5 Formal proof of non-bypassable instrumentation

The previous section discusses that the safety of instrumentation by listing the properties enforced and potential attacks defeated. This section pursues a formal proof of instrumentation non-bypassability. At a high level, the properties enforced by the core techniques in previous sections can be summarized as follows:

- "What You Disassemble Is What You eXecute (WYDIWYX):" An instrumented program will only execute code that was successfully disassembled *and* instrumented.
- Disassembly errors will not lead to undefined program behavior (although errors in added instrumentation can).
- Instrumentation cannot be bypassed, nor can the control flow within the added instrumentation be subverted.

We make these properties more precise and establish them in this section. We begin by making the following observation about the instrumentation scheme described in the preceding section:

Observation 2.2. After the above instrumentation, the only ICF transfers left in the program are those in the address translation trampoline: rest of them have been translated into direct control-flow transfers using labels of assembly statements in the instrumented program. Moreover, the only ICF targets are the entries in the hash table.

We need a few definitions to formalize the notion of sound policy enforcement.

Definition 2.3 (Code). Code is a sequence of tagged instructions. An instruction tagged *valid* contains a valid machine instruction, whereas an instruction tagged *invalid* may contain an arbitrary sequence of bytes. An instruction may optionally be associated with a unique label that can be used as a control-flow target.

An mislabeled instruction can be used to represent any sequence of data bytes. For instructions produced by a disassembler, we used the location of each instruction to generate a corresponding unique label as described earlier.

Definition 2.4 (Code Instrumentation). A code instrumentor `I` takes a piece of code C and produces another piece of code $C' = \mathtt{I}(C)$. For each instruction $i \in C$, the instrumentation code is mapped to a code sequence s consisting of three parts: $pre(i)$, i' and $post(i) \in C'$. In particular, $pre(i)$ and $post(i)$ represent the instrumentation code inserted before and after the original instruction. Both of them can be empty. i' is the corresponding instruction of i. I^{-1} generates a one-to-one mapping of the beginning of s and i, where each s maps to one i'. Note that any other instruction addresses in s except the beginning will not have such a mapping to any instruction in C.

Note that an instrumentation should have other important properties, e.g., As for problem behavior is concerned, C' should be semantically equivalent to C on inputs of interest, while on some invalid or undesired inputs, C' may terminate before C. This divergence of behavior on undesired inputs corresponds to the enforcement of a security policy by the instrumentor. While these aspects of instrumentation are helpful in understanding them, we do not formalize them since they are not needed in the discussion below. In particular, based on the above the following property is defined:

Definition 2.5 (Instruction-boundary enforcement (instr-CFI)). An instrumentation $\mathtt{I}$ is said to ensure instr-CFI if, for every control flow instruction i' that corresponds to an original instruction $i \in C$, the target d is constrained as follows:

1. d is the beginning of a s in $\mathtt{I}(C)$
2. if $I^{-1}(i)$ is defined, then i must be a direct control-flow transfer, and $I^{-1}(d)$ must also be defined.

The first condition is obvious: it captures the essence of instr-CFI, i.e., the instrumented code does not jump to the middle of instrumentation code sequence. The second condition goes beyond this simple requirement, and is intended to ensure that newly added instrumentation won't be compromised by the original code. Since the ability of original code to subvert newly added instrumentation is the only concern, the second condition applies only to instructions that are from the original code. It states that these instructions cannot be used to jump to any of the instructions newly added by the instrumentation. This ensures that the original code has no ability to interfere with the logic of the instrumentation. To illustrate this point, consider an instrumentation that sandboxes a register value by performing a bitwise-and operation with a constant. The second condition in the definition rules out the possibility that a control transfer in the original program can skip over this sandboxing operation by jumping past it.

Lemma 2.6. The instrumentation described in Section 2.3.3 ensures instr-CFI.

Proof: The first condition holds because, as stated in Observation 2.2, the only ICF transfers are in the address translation trampoline, and all of these transfer to legitimate instruction boundaries in the hash table. Moreover, direct flow transfers cannot jump to the middle of instructions because they use labels, which are associated only with the beginning of instructions.

With respect to the second condition, from Observation 2.2, the only ICF transfers are in trampoline code that is newly added by the instrumentation. Being newly added, I^{-1} is null for this instruction. This establishes the first part of the second condition. Moreover, since all direct control flow transfers are transformed to use labels of the form `L_`A, it goes without saying that the second part of the condition holds as well. ■

It is now ready to establish that disassembly errors will not cause execution of unrecognized or uninstrumented code.

Corollary 2.7. Disassembly errors do not compromise policy enforced by instrumentations. In particular, disassembly errors:

- do not lead to execution of unrecognized or uninstrumented code
- do not enable control-flow transfers into the middle of instrumentation code.

The first of these two properties is referred as "What You Disassemble Is What You eXecute (WYDIWYX)." It provides a sound basis for static analysis and instrumentation of binaries: the results computed will remain sound even if there were errors in disassembly. The second condition is motivated by the fact that jumps into the middle of instrumentation can defeat or bypass inline checks, and hence violate the policy intended by the instrumentation.

Proof: Note that all direct control-flow transfers go to legitimate labels (and hence legitimate instructions) in the instrumented program. From Observation 2.2, it follows that all ICF transfers go to legitimate code in the hash table. The second condition follows from the second condition in the instr-CFI property definition.■

We now turns the focus on functionality, i.e., the potential for disassembly errors to break the instrumented program. Disassembly errors can fall into three categories:

- *Failure to disassemble legitimate code:* This error isn't possible with linear disassembly, since the entirety of all code segments is disassembled.[10]
- *Disassembly of non-code,* e.g., disassembling of data stored within code segment.
- *Incorrect disassembly of legitimate code:* This can be further subdivided into:
  - *decoding error,* i.e., incorrect decoding of a legitimate instruction.
  - *instruction boundary error,* i.e., an attempt to decode from the middle of a legitimate instruction.

While decoding errors are possible, they represent simple implementation bugs that can usually be found and fixed easily. So, henceforth, we only consider disassembly of non-code and instruction boundary errors, and make a few observations about instrumentation correctness despite these errors.

Observation 2.8. Disassembly of non-code, on its own, does not break the functionality of instrumented application.

Proof: If non-code is disassembled and instrumented, then the result is not meaningful code. However, this does not cause any problems since the program, while exercising its normal functionality, will never jump into invalid code. However, program may access this non-code as data. Such accesses go to the original code, this is because instrumentations ensure transparency, meaning that data address calculation including those with code pointer involved will result in the same values as the original program does. This is ensured by the transparent instrumentations of control flow transfer instructions (Details in Section 2.3.3). Since original code is left in its place, instrumented code will read the exact same data as the original program, and hence will function the same as before. ■

Observation 2.9. Disassembly errors will never lead to undefined behavior: they can cause premature termination of instrumented program, but this will occur before any incorrectly disassembled code is executed.

[10]Strictly speaking, the disassembly approach can mark regions as non-code, and this raises the possibility of failure to disassemble legitimate code. However, note that regions are marked as non-code only when disassembly errors are detected, which implies that the region contains non-code. Moreover, code pointer analysis and data region recognition algorithms designate the smallest possible region as non-code, thus avoiding the possibility of leaving out legitimate code from disassembly.

Proof: We have already shown that disassembly of non-code does not affect correctness, so only instruction boundary errors need to be considered. Note that execution starts at the entry point specified in the ELF-header. Since linear disassembly starts from this point, there will be no instruction boundary errors at the beginning of program execution. Starting from the entry point, it is reasonable to inductively argue that instruction boundary errors cannot occur in the instructions that are executed next, until a control-flow transfer is taken. Consider the first control-flow transfer where an instruction boundary error occurs. Since the program's behavior so far has not been affected by errors, the control-flow target represents valid instruction. An instruction boundary error at this target means that it falls in the middle of a disassembled instruction. Because of the error checks performed during disassembly, this is not possible for a direct control-flow transfer. For an ICF transfer, since the address trampoline code checks that the target corresponds to the beginning of a disassembled instruction, this control transfer will be disallowed. Thus, program execution is aborted before executing any incorrectly disassembled code. ■

2.6 Summary

In this chapter, we have presented the core techniques of static binary instrumentation used in PSI. In particular, we have discussed static binary disassembly and code pointer analysis. Based on these techniques, we discussed the instrumentation techniques using address translation. We further elaborated it with instrumentation optimizations afterwards. In the following section, we have discussed control flow integrity that is built on top of address translation. We have showed how to build a compatible but strong CFI policy on complex COTS binaries. After that, we prove that our instrumentation incorporated with CFI is non-bypassable.

Chapter 3

Platform Implementation, API and Evaluation

The focus of this chapter is to discuss the implementation details of PSI, its instrumentation API for arbitrary instrumentation and a systematic evaluation.

The ability to support generic instrumentation is based on two main observations: (a) PSI avoids inline instrumentation and thus the instrumented code does not suffer from the size limitation, i.e., the instrumented code could be in any length; (b) the instrumented code address is mapped to original code address using address translation, i.e., all indirect control transfers will be correctly handled regardless of instrumentation.

3.1 System implementation

PSI instruments executables as well as all of the shared libraries used by them. On each invocation, PSI takes a binary file (executable or library) as input, and outputs an instrumented version of this binary. This invocation may occur before execution, or on-the-fly during program execution.

Figure 4.1 shows the architecture of PSI. It consists of three main components: a binary analyzer, an instrumentor and a binary generator. The core techniques in binary analyzer have been discussed in Section 2.1 and Section 2.2. It takes a binary as input, disassembles it, and then constructs a control-flow graph (CFG). This CFG is then sent as the input for static analysis. The instrumentor mentioned

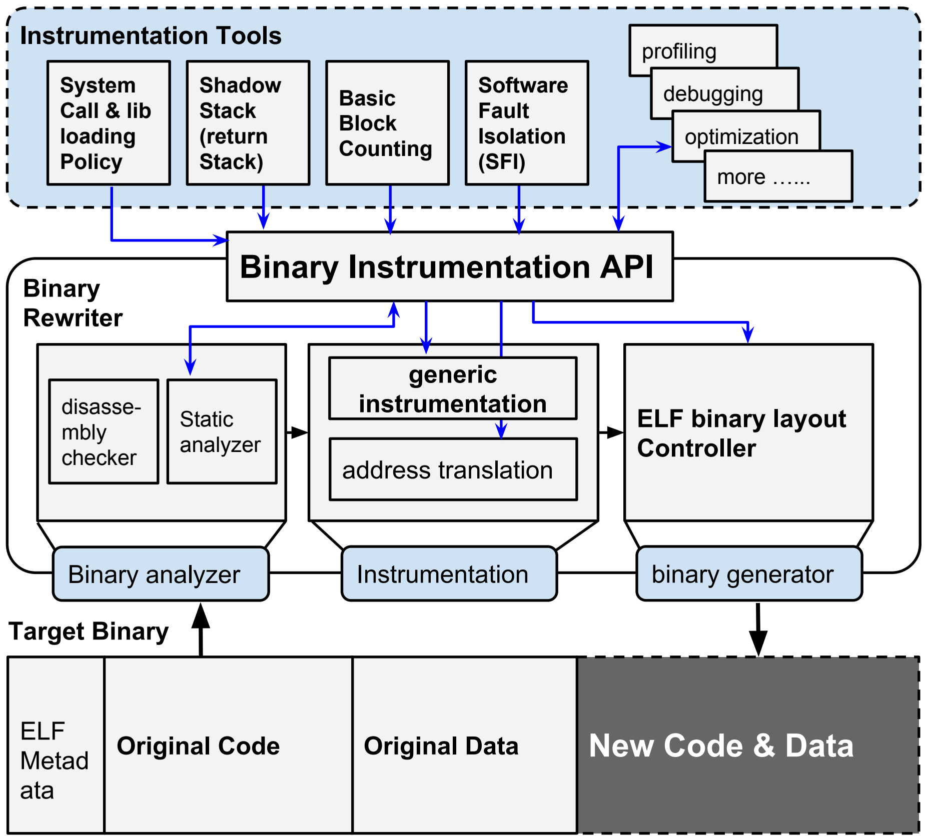

FIGURE 3.1: Architecture Overview of PSI

in Section 2.3 will take the analyzed disassembly, performs instrumentation and output new assembly code. Note that control flow integrity policy mentioned in Section 2.5 will be taken as the mandatory instrumentation. Once instrumentor is done, the binary generator will re-assemble the code into an object file, fixing all the relocations and inject the new code back into original binary.

The heart of PSI is the instrumentor, which provides an expressive API for static binary instrumentation. Two levels of API are supported. The low-level API operates at the instruction level, and supports operations for inserting assembly language snippets at desired points. The high-level API allows insertion of calls to instrumentation functions that may be written in a language such as C. This code is compiled into a shared library, and this platform ensures that these functions can be called in a secure manner from (and only from) the calls inserted using the high-level instrumentation API. This API is further described in Section 3.2.

The actual task of instrumentation is performed by *instrumentation tools*, which are programs that use the API provided by the platform to instrument applications and the libraries used by them.

The API provided by PSI not only allows tools to control the instrumentation phase, but also other phases such as static analysis, address translation, etc.

3.2 Instrumentation API

Our PSI platform provides a simple API for custom instrumentation of binaries. Being a static rewriting tool, all instrumentation operations occur in one shot on PSI: the entire CFG for a binary is presented to the code instrumentor, which traverses the CFG and adds all the desired instrumentation. This contrasts with DBI tools where instructions and basic blocks are discovered one by one, just before their first execution, and the code instrumentor invoked separately on the newly discovered basic block.

After disassembly, PSI constructs a control-flow graph (CFG) of the program. The nodes in the CFG are basic blocks, each of which consists of a sequence of instructions. All incoming control transfers into a basic block go to its first instruction, while all control transfers out of the block occur on its last statement. Note that every indirect control flow target computed by the static analysis is considered in defining these basic blocks. Since this analysis estimates a superset of possible indirect targets, the basic blocks computed by the instrumentation code can be smaller than those computed by a compiler.

The entire CFG can be accessed using the API function `getCFG`, while the list of all basic blocks and instructions can be obtained using the functions `getBBs` and `getInsns`. The API also provides operations to iterate through instructions and basic blocks in a CFG, and instructions in a basic block. Also supported are operations to examine instructions. These operations are based on the Intel's xed2 instruction encoder/decoder library. Some of the most commonly used operations are `isCall`, `isRet`, `isTest`, `isSysCall`, `isMemRead`, `isMemWrite`, `getTarget`, and `getSrc`.

3.2.1 Insertion of assembly code snippets

Instrumentation can be performed at a low or high level. At the low level, an instrumentation snippet is inserted as follows:

$$\texttt{ins\_snippet}(target, location, snippet)$$

Here, *target* is a reference to an instruction or a basic block, and is specified using a reference to the corresponding object, or by using a label. The parameter *location* is one of `BEFORE` or `AFTER`, and snippet is a string consisting of assembly code that is to be inserted. For call instructions, one extra location `AFTER_CALL` is defined. To ensure transparency of return addresses on the stack, a call is translated into a push instruction that pushes the original return address, followed by a jump that transfers control to the target function. `AFTER_CALL` corresponds to the point between push and jump.

Instead of inserting additional instrumentation, some applications may require replacement of existing instructions. This is done using the following API function:

$$\texttt{replace\_ins}(target, new\_snippet)$$

Instruction emulation is a purpose for which this API function comes handy: we replace the original instruction with a snippet that emulates it.

PSI provides a private thread local storage (TLS) area that can be used by assembly snippets to store their data. This private TLS, which is independent of the one provided by `glibc`, is organized into two arrays `TS` and `GS` that are both initialized with all zeros. The size of these arrays is configurable, but they default to one memory page. Snippet code can use the identifiers `TS_`n and `GS_`n to access the nth word of the arrays `TS` and `GS` respectively.

3.2.2 Insertion of calls to instrumentation functions

The benefit of the snippet API is that it can be more efficient since the instrumentation writer can minimize the number of instructions executed. Its downside is that instrumentation has to be performed in assembly, and that it is more complex. In contrast, the higher level API simplifies instrumentation but is generally less efficient. It enables the insertion of calls to handler functions in a shared library defined by the instrumentor. Several low level details are handled automatically by the high-level API. These include saving/restoring of registers and flags, switching to a different stack, resolving the symbolic name of the user-defined handler function, making the program state available through a high-level data structure called `Context`, and so on. These factors simplify the instrumentation task, and allows the handler code to be written in higher level languages (currently, C/C++). This API is accessed using the following function:

$$\texttt{ins\_call}(target, location, name, args)$$

Here, *target* and *location* have the same meaning as the snippet API. The name of the function to be invoked is specified using the string parameter *name*. This function should have the following prototype:

$$\texttt{void handler}(\texttt{struct Context } *\texttt{c}, \ldots)$$

It takes a first parameter that represents the runtime context of the instrumented program, including all of the CPU registers, stack, etc. The subsequent parameters are exactly those that were included in the *args* parameter to i*ns_call*.

3.2.3 Controlling address translation

As described earlier, address translation instrumentation, which regulates ICF transfers, is automatically added by PSI. PSI provides some API functions so that an instrumentation developer can exercise finer control over ICF transfers. These functions can be used by an instrumentation tool that implements a more sophisticated ICF target analysis to further restrict indirect branches. Even without performing more static analysis, an instrumentation tool may enforce a more restrictive policy, e.g.,

- all returns should go to instructions following calls
- (some or all) indirect jumps should not target addresses outside the current module

These restrictions can be specified using the API function:

$$\texttt{rm\_indirect\_target}(src\_addrs, target\_addrs)$$

The argument src_addrs is a list of the labels of the ICF transfer instructions whose targets should be restricted. If it is empty, then the operation is applied to all ICF transfer instructions in the module. The second argument is also a list of labels,which represents the list of valid targets.

Custom address translation instrumentation can also be used to relax a previously specified policy. This is done using the following API function:

$$\texttt{add\_indirect\_target}(src\_addrs, target\_addrs)$$

The platform will keep track of possible targets for each source address, and will generate a unique address translation trampoline for each set of source addresses that share the same set of possible targets.

3.2.4 Runtime event handling

Finally, the API supports registration of instrumentation functions that will be called when certain events occur at runtime, such as program/thread startup or exit, loading of libraries, and system calls:

- `register_pre_syscall_handler()`
- `register_post_syscall_handler()`
- `register_library_load_handler()`
- `register_thread_start_handler()`
- `register_thread_terminate_handler()`
- `register_program_start_handler()`
- `register_program_terminate_handler()`

These API calls take a function name as their argument.

3.2.5 Development of instrumentation tools

To develop an instrumentation tool, user provides the tool code, and optionally, a *client library.* The tool code uses the API provided by PSI to realize an instrumentation tool. Tools that use the high-level API need a mechanism to provide the definitions of function calls inserted using that API. This is the role of the client library.[1]

The instrumentation tool code is written in `perl` and should be stored in a source file, say, `bbcount.pl`. To instrument a binary file `xyz`, the tool developer invoke PSI instrumentor and take the instrumentation tool as a parameter. Specifically, the following sequence of commands is used for developing an instrumentation tool using the low-level API:

```
psi_instrumentor -t bbcount.scpt [-m map] -o instrumented_xyz -- xyz
```

If a high-level API is used, the client library, say `bbcount.c` needs to be compiled first into a shared library `bbcount.so`. At runtime, calls made by instrumentation to client library functions in `bbcount.so` need to be resolved. In principle, resolving a client function name is straight-forward: use the standard C-compiler to produce a shared library from the client library source, and include this library in the dependency list for the instrumented binary. However, such an approach will violate the security requirement because client library functions would be callable by application code. Instead, PSI restricts these functions to be callable only from the added instrumentation. To ensure the restriction, we have developed a dedicated symbol resolution technique for resolving function names in the client library.

To resolve client functions, the basic idea is to create a global address table (GAT) that contains the memory locations where the client functions have been loaded. To simplify the look-up process, PSI translates function names in the client library to integer indices. This enables GAT to be a simple array indexed by these integer indices. A mapping file specifying a name-to-index mapping is generated during the compilation of the client library:

```
   psic -o bbcount.so bbcount.c

   psic --extract-map -o bbcount.map bbcount.so
```

[1]Note that the tool code is used at the time of instrumenting a binary, whereas the client library functions are used during the execution of the instrumented applications. This is why tool code is separated from the client library.

Now, this mapping file will be used by the instrumentor to translate calls to client library functions made in the instrumentation code. The mapping file is also used by a modified loader, which uses this mapping to populate the `GAT`. Specifically, for each function f defined in the client library, the loader finds its index i_f from the mapping file, and stores the location where f is loaded in `GAT`$[i_f]$. After populating `GAT`, it is made read-only. In addition, none of the locations in the client library are ever added to the translation tables. These two steps ensure that code in the client libraries can never be invoked by the original code.

Resolution of the variable names could be accomplished in a similar way with the help of another GAT and a corresponding name-to-index mapping.

Finally, user could launch the translated binary using the following command:

```
psi_loader -t bbcount.so  -- instrumented_xyz
```

3.3 On-demand instrumentation

One of the drawbacks of a purely static instrumentation approach is that the user has to compute the list of all shared libraries that may be used when an instrumented program is run, and create instrumented version of these libraries. This is a difficult task, since some libraries may be loaded long after program execution begins. Some of these libraries may reside at user-specified locations known only at runtime. In order to support seamless instrumentation of such libraries, our PSI platform provides an option to generate instrumented libraries on-the-fly. In particular, PSI leverages a modified the loader to support an option to specify a configuration file that is consulted when an instrumented application requests to load an uninstrumented library. (If this option is not specified, then any request to load an uninstrumented library will be denied.) This configuration file must specify the name of the libraries containing the tool code, client library code, and the mapping file. The loader will then invoke PSI to create an instrumented version of the uninstrumented library, and then load this version.

Instrumented libraries are stored in a disk cache for subsequent uses. This cache can store multiple versions of the same library, each corresponding to a different tool.

In principle, PSI could be deployed on a system-wide basis, and use a shared cache across all users. However, currently PSI relies on a simpler scheme that uses a per-user cache. The cache is simply a directory owned by the user, say, `${home_psi}/bob/`.

When the loader is asked to load a library, say, `/usr/lib/abc.so` by a process instrumented with a tool `bbcount` and owned by `bob`, the loader concatenates the library name to the cache location, i.e., looks for the file:

```
${home_psi}/bob/bbcount/usr/lib/abc.so
```

If found, this file is loaded. If not, the loader invokes `psi` to instrument `/usr/lib/abc.so` with the tool `bbcount` and stores this result in the cache, and loads the instrumented version from the cache.

Note that libraries with the same instrumentation but with different compilation options or client libraries may cause compatibility problems. To avoid these, the loader checks the library version, compilation options, as well as the client library version. It also checks the timestamps on the tool code and client library code, and if they are newer than the cached version, then a new, instrumented version is generated and the copy in the cache is updated.

On-demand instrumentation can be applied to executables as well, and serve to support seamless instrumentation of applications that involve running multiple executables

3.4 Illustrative instrumentation examples

This section illustrates the API described in the preceding section. This section will use several examples to illustrate the flexibility, versatility and the ease-of-use benefits of PSI.

3.4.1 Basic block counting

Basic block counting has been used to illustrate previous DBI tools such as Pin and DynamoRIO. Moreover, an optimized version of this tool is available for these platforms, thus providing a good basis for performance comparison. For this reason, we illustrate the platform and API using this example. (See Figure 3.2.) The core of the instrumentation is to increment a memory location. However, since this operation affects CPU flags, it is necessary to save and restore them. This is performed in the snippet *unopt*.

It would be safe to avoid flag save/restore, and use the optimized snippet *opt*, if the flags aren't live at the snippet insertion point. To simplify the example, we avoid a general liveness analysis. Instead, it searches two common instances of instructions that set the flags, namely, `test` and `cmp`, and insert the increment instruction just before them.

For brevity, the author has omitted the code for printing results. This can be achieved by registering a thread termination handler that will accumulate the count from thread-local TS_1 into a global location, say, GS_1. Finally, a program termination handler needs to be registered that prints the value of GS_1.

```
unopt = "mov%eax, TS_0;
         lahf;
         incl  TS_1;
         sahf;
         mov TS_0, %eax"
opt = "incl TS_1"
foreach bb in getBBs() {
   found = false
   foreach insn in bb {
      if isTest(insn) or isCmp(insn) {
         found = true
         ins_snippet(insn, BEFORE, opt)
         break
      }
   }
   if !found
      ins_snippet(bb, BEGIN, unopt)
}
```

FIGURE 3.2: An instrumentation tool for Basic Block Counting

3.4.2 System call policy enforcement

System call policy enforcement is a well-known protection technique for sandboxing. PSI provides a simple API to enforce system call policies. Performance overheads are minimal, comparable to library interposition. At the same time, it provides security comparable to ptrace, a much heavier-weight mechanism used in tools such as `strace`.

System call policy enforcement is implemented by registering system call event handlers `register_pre_syscall_ handler`() and `register_post_syscall _handler`(). This platform is able to identify system calls that use `int 0x80` mechanism as well as the faster method that uses the `sysenter` instruction. The handler function can use its `Context` argument to determine system call arguments, which are stored in registers. The handler can examine and/or modify these arguments.

3.4.3 Library load policy enforcement

Malicious library loading is a well known strategy employed by security exploits to circumvent injected code defenses such as those that prevent execution of data.

A library loading policy is implemented using a tool that registers a handler for the event `register_library_load_ handler`(). The handler can then examine the name of the library being loaded, and disallow it if need be. In this tool, rather than enforcing a policy, the developer simply logged a message that can be processed subsequently to identify how many libraries are loaded by an application, and what fraction of them are loaded. This tool has been used on a collection of commonly used command-line and GUI applications and a significant fraction of libraries (specifically, over 40% in this experiment) have been found loaded after the commencement of application execution.

3.4.4 Shadow stack

Shadow stack [115] is a well-known technique for defending against return address corruption. The idea is to maintain a second copy of return addresses on a "shadow" stack, and check the two copies for consistency before each return. Successful exploits now require both copies of the return address to be compromised, which is harder than circumventing protection mechanisms such as stack canaries.

Binary-based return address defender [115] was the first to use binary instrumentation to implement shadow stacks. It inserts additional code at function prologue and epilogue to respectively push and check return addresses on shadow stack. While their approach is useful against buffer overflow attacks on return addresses, they are not effective against ROP attacks that mainly use unintended return instructions, as there will be no shadow stack checks preceding such “instructions.” Note that the initial exploit can be triggered without compromising a return address, e.g., by corrupting a function pointer.

ROPdefender [63] addresses this weakness using DBI. As DBI techniques ensure instrumentation of all code before execution, their approach will instrument unintended returns as well, and hence prevent ROP attacks. We compare the performance of their implementation, which was based on Pin, with PSI. For this purpose, a shadow stack instrumentation tool is developed as shown in Figure 3.3. Its implementation emphasizes ease of development and compatibility with legacy software, and there is no significant effort done to optimize it. Thus, the performance results reflect the performance strengths of PSI, rather than the efficiency of the instrumentation tool.

Note that being a static instrumentation technique, PSI will not instrument unintended return instructions. However, the runtime checks on indirect targets will stop any attempts to jump to such instructions. Moreover, attacks aimed at evading shadow stack checks, such as those based on jumping into the middle of (or past the end of) checking code will be defeated as well.

In Figure 3.3, the shadow stack could be initialized at the time a new thread is spawned. However, we opted for a simpler (but less efficient) approach where the validity of shadow stack is checked on each call instruction, using *chk_init_shadowstk*. This snippet uses another support function to allocate a shadow stack if it is not already set up. Once the shadow stack is in place, `push_shadowstk` is used to push a copy of the return address to the shadow stack.

Checking the integrity of returns is more complex, so we use a high-level function to perform this action. Note that uses of `longjmp` can cause a mismatch between shadow and main stack. This occurs because stack frames have been popped off the main stack. The solution to this problem, used in previous works [63, 115], is to successively pop off entries from the shadow stack until the two match. However, if the bottom of shadow stack is reached, that implies an attack, and the program is aborted.

```
/* shadow stack pointer is stored in TS_2 */
chk_init_shadowstk= "
            cmp  $0x0, TS_2;
            jnz  L001;
            call $alloc_stack;
        L001: ";
push_shadowstk = "
            mov  %eax, TS_0; mov %ebx, TS_1;
            subl $4, TS_2;
            mov  TS_2, %eax;
            mov  (%esp), %ebx; mov %ebx, (%eax)
            mov  TS_0, %eax; mov TS_1, %ebx;"
check_return(Context*ctxt) {
    shadow_sp = ctxt->TS[2]
    ret = getmem(ctxt->ESP)
    while !empty(shadow_sp)
      if (pop(shadow_sp) == ret) {
        ctxt->TS[2] = shadow_sp
        return
      }
    abort()
}
foreach insn in getInsns()
    if isCall(insn) {
      ins_snippet(insn, BEFORE, chk_init_shadowstk)
      ins_snippet(insn, BEFORE, push_shadowstk)
    }
    else if isRet(insn)
      ins_call(insn, AFTER_CALL, check_return)
```

FIGURE 3.3: Shadow Stack Defense

As noted by the authors of ROPdefender, real-world programs introduce a few benign violations of shadow stack checks, and these need to be handled. We already described how violations due to `longjmp` are handled. Other violations occur due to lazy binding used by the dynamic loader, the occurrence of C++ exceptions, UNIX signals, and System V thread context switches due to functions such as `setcontext` and `getcontext`. The core idea used in ROPdefender is that of recognizing which return instructions in the binary cause these exceptions (each of them occur within a specific routine in the loader or libc), and modifying the instrumentation of those instructions. We used the same idea in the implementation, but have omitted the details to conserve space.

3.5 System Evaluation

In this section, we present a comprehensive evaluation of PSI for the purpose of answering the following questions:

- *Does PSI work on COTS binaries?*
- *How does PSI compare with defenses against control flow hijacking attacks?*
- *How efficient PSI is compared with DBI systems?*

The evaluation in this chapter targets the following aspects: (a) functionality, (b) security, (c) runtime performance and (d) comparative analysis with dynamic binary instrumentation tools.

Functionality evaluation is focused on two aspects: (a) consistency of program semantics between instrumented application and original application and (b) correctness of binary disassembly. To evaluate semantic consistency, we tested our instrumented programs including the SPEC 2006 benchmark and several real world applications most frequently used on Linux. No runtime error was found in the testing evaluation illustrated in Section 3.5.1.1. Since testing explores only a fraction of program paths, we undertook a more complete evaluation of disassembly correctness as described in Section 3.5.1.2.

In terms of security, we first evaluate the strength of its CFI with previous work using the AIR metric proposed in Section 2.5.3. In addition, we evaluate PSI's ability of security defense in Section 3.5.2.1. Overall, the results in Section 3.5.2 illustrate that PSI is capable of defeating vast majority of control flow hijacking attacks and mitigating code reuse attacks. In particular, 93% of ROP gadgets [2] have been eliminated.

Performance of PSI is evaluated in Section 3.5.3. In this chapter, we evaluate both runtime overhead and space overhead. The results demonstrate that PSI has comparable baseline performance with DBI tools such as DynamoRIO and Pin. For some instrumentations such as shadow stack, PSI achieves much better performance.

Finally, a comparative analysis is performed over DBI tools such as DynamoRIO and Pin. In particular, four aspects have been extensively evaluated including:

[2] A gadget refers to a sequence of instruction ending in an indirect branch.

Application Name	Experiment
Wireshark v1.6.2	capture packets on LAN for 20 minutes
gedit v3.2.3	open multiple files; edit; print; save
lyx v2.0.0	open a large report; edit; convert to pdf/dvi/ps
acroread9	open 20 pdf files; scroll;print;zoom in/out
mplayer 4.6.1	play an mp3 file
firefox 5 (no JIT)	open web pages
perl	execute a complex script, compare the output
vim	open file, copy/paste, search, edit
gimp-2.6	load jpg picture, crop, blur, sharpen, etc.
lynx 2.8.8dev	open web pages
ssh 5.8p1	login to a remote server
evince 3.2.1	open a large pdf file

FIGURE 3.4: Functionality Tests

baseline performance, instrumentation application performance, micro-benchmark performance and performance on real world program benchmarks. Evaluation of this part is thoroughly discussed in Section 3.5.4

3.5.1 Functionality

3.5.1.1 Testing transformed code

We verified the runtime correctness of SPEC CPU2006 programs as illustrated in Figure 3.6. Note that this benchmark comes with scripts to verify outputs, thus simplifying functionality testing.

In this work, we also tested many real world programs including coreutils-8.16 and binutils-2.22, and medium to large programs such ssh, scp, wireshark, gedit, mplayer, perl, gimp, firefox, acroread, lyx as well as all the shared libraries used by them including `libc.so.6`, `libpthread.so.0`, `libQtGui.so.4`, `libQtCore.so.4`. The experiment results are shown in Figure 3.4.

Altogether, 786 shared libraries were transformed during the experiments. The total size of code transformed was over 300 MB, of which size of the libraries was about 240MB. We tested each of these programs to check that they worked correctly. A subset of these tests is shown in Figure 3.4.

3.5.1.2 Correctness of disassembly

Since testing explores only a fraction of program paths, we undertook a complete evaluation of disassembly correctness, i.e., verifying consistency of disassembly and compiler generated assembly. To obtain the assembly code generated by the compiler, we recompiled several large programs, including Firefox 5, GIMP-2.6 and glibc-2.13. Specifically, the option `--listing-lhs-width=4 -alcdn` of GNU assembler was turned on to generate listing files containing both machine code and assembly. The content of these files was then compared with disassembly.

Note that multiple object files are combined by the linker to produce an executable or a library. We intercepted the linker `ld` to associate instruction locations in the binary with those in each object file. This information was then used to generate a mapping between compiler-produced assembly snippets in each object file and the corresponding parts of the disassembly code..

Module	Package	Size	# of Instructions	# of Errors
libxul.so	firefox-5.0	26M	4.3M	0
gimp-console-2.6	gimp-2.6.5	7.7M	385K	0
libc.so	glibc-2.13	8.1M	301K	0
libnss3.so	firefox-5.0	4.1M	235K	0
libmozsqlite3.so	firefox-5.0	1.8M	128K	0
libfreebl3.so	firefox-5.0	876K	66K	0
libsoftokn3.so	firefox-5.0	756K	50K	0
libnspr4.so	firefox-5.0	776K	41K	0
libssl3.so	firefox-5.0	864K	40K	0
libm.so	glibc-2.13	620K	35K	0
libnssdbm3.so	firefox-5.0	570K	34K	0
libsmime3.so	firefox-5.0	746K	30K	0
ld.so	glibc-2.13	694K	28K	0
gimpressionist	gimp-2.6.5	403K	21K	0
script-fu	gimp-2.6.5	410K	21K	0
libnssckbi.so	firefox-5.0	733K	19K	0
libtestcrasher.so	firefox-5.0	676K	17K	0
gfig	gimp-2.6.5	442K	17K	0
libpthread.so	glibc-2.13	666K	15K	0
libnsl.so	glibc-2.13	448K	15K	0
map-object	gimp-2.6.5	257K	15K	0
libresolv.so	glibc-2.13	275K	13K	0
libnssutil3.so	firefox-5.0	311K	13K	0
Total		58M	5.84M	0

FIGURE 3.5: Disassembly Correctness

Figure 3.5 illustrates results of the disassembly testing. About 58MB of executable files including code and data, corresponding to a total of about 6M instructions, have been tested, with no error reported.

3.5.1.3 Testing code generated by alternative compilers

We also applied the instrumentation to two programs compiled using LLVM. In particular, we use Clang 2.9 to compile two programs in the `OpenSSH` project, `ssh` and `scp`. Experiments illustrate that LLVM generated binaries function correctly when they are used to login to a remote server and copy a large file to/from the server.

We noticed that the padding used by LLVM compiler was quite different from that of gcc, but the implementation was able to handle it without any changes.

In the experiments, we discovered that padding style is a very obvious difference between LLVM generated binary and gcc and generated binary. LLVM choose to use some undocumented x86 instructions to perform a padding that requires size over 8. In contrast, gcc uses several shorter instructions to perform the padding.

3.5.2 Security evaluation

PSI has integrated a coarse grained CFI policy, BinCFI. The strength and effectiveness of this CFI design is important for instrumentation tools built on top of it. This section will focus on security evaluation of the CFI property that PSI has integrated.

3.5.2.1 CFI effectiveness evaluation

To evaluate the effectiveness of this CFI property, we proposes a metric called Average Indirect target Reduction (AIR) defined in Section 2.5.3.

Figure 3.6 compares the AIR metric of BinCFI with that of strict-CFI, reloc-CFI, bundle-CFI and instr-CFI proposed in Section 5.3.3. To calculate AIR metric of reloc-CFI, we recompiled SPEC2006 programs using “-g” and a linker option “-Wl,-emit-relocs” to retain all the relocations in executables.

To calculate AIR of bundle-CFI, SPEC2006 is recompiled using Google native client provided gcc and g++ compilers. Since bundle-CFI restricts ICF targets to 32-byte boundaries, 31/32 of the compiled binary code is eliminated as ICF targets. However, the value of AIR is smaller because the base is the original program size; programs compiled using Native Client tool-chain are larger in size due to reasons such as the need to introduce padding to align indirect targets at 32-byte boundaries.

Name	Reloc CFI	Strict CFI	**Bin CFI**	Bundle CFI	Instr CFI
perlbench	98.49%	98.44%	**97.89%**	95.41%	67.33%
bzip2	99.55%	99.49%	**99.37%**	95.65%	78.59%
gcc	98.73%	98.71%	**98.34%**	95.86%	80.63%
mcf	99.47%	99.37%	**99.25%**	95.91%	79.35%
gobmk	99.40%	99.40%	**99.20%**	97.75%	89.08%
hmmer	98.90%	98.87%	**98.61%**	95.85%	79.01%
sjeng	99.32%	99.30%	**99.10%**	96.22%	83.18%
libquantum	99.14%	99.07%	**98.89%**	95.96%	76.53%
h264ref	99.64%	99.60%	**99.52%**	96.25%	80.71%
omnetpp	98.26%	98.08%	**97.68%**	95.72%	82.03%
astar	99.18%	99.13%	**98.95%**	96.02%	78.00%
milc	98.89%	98.86%	**98.65%**	96.03%	79.74%
namd	99.65%	99.64%	**99.59%**	95.81%	76.37%
soplex	99.19%	99.10%	**98.86%**	95.50%	77.37%
povray	99.01%	98.99%	**98.67%**	95.87%	78.03%
lbm	99.60%	99.50%	**99.46%**	96.79%	80.92%
sphinx3	98.83%	98.80%	**98.64%**	96.06%	80.75%
average	*99.13%*	*99.08%*	**98.86%**	*96.04%*	*79.27%*

FIGURE 3.6: AIR metrics for SPEC CPU 2006.

The results in Figure 3.6 demonstrate that BinCFI maintains a good strength compared with reloc-CFI and strict-CFI. However, neither reloc-CFI or strict-CFI support large and complex COTS binaries.

3.5.2.2 Control-Flow hijack attacks

To evaluate the effectiveness of BinCFI against control flow hijack defense, we used RIPE [143] exploit suite. RIPE is a benchmark consisting of 850 distinct exploits including code injection, return-to-libc and ROP attacks. RIPE illustrated these attacks by building vulnerabilities into a small program. Exploit code is also built into this program, so some of the challenges of developing exploits, e.g., knowing the right jump addresses, are not present. As such, techniques such as ASLR have no impact on RIPE. So, the only change that can be experimented with is enabling or disabling data execution prevention (DEP)[3].

[3]DEP is a security feature that supported by modern CPUs to prevent execution of data as code.

	DEP disabled	DEP enabled
Original	520	140
CFI	90	90

Figure 3.7: Security Evaluation using RIPE

Originally, on Ubuntu 11.10 platform, 520 attacks survived with DEP disabled. With DEP enabled, 140 attacks survived. All of these attacks are return-to-libc attacks.

The 2nd row in Figure 3.7 shows BinCFI defeated 430 attacks including 380 code injection attacks and 50 return-to-libc attacks, even when DEP is disabled. In both scenarios, when DEP was enabled or disabled, there were 90 function pointer overwrite attacks that survived in CFI.

Code injection attacks were defeated by CFI because global data, stack and heap are prohibited targets of ICF transfers. 50 out of 140 return-to-libc attacks were defeated because they overflowed return addresses and tried to redirect control flow to the libc functions and thus violate the policy of BinCFI.

The success of these attacks is some what of an artifact of RIPE design that includes exploit code within the victim program. Since pointers to exploit code are already taken in the program, they are identified as legitimate targets and permitted by the approach. If the same attacks were to be carried out against real programs, only a subset of them will succeed: those that overwrite function pointers with pointers to other local functions. In this subset of cases, previous CFI implementations (although not necessarily their formulations) would fail too, as they too permit any indirect call to reach all functions whose address are taken.

3.5.2.3 ROP attacks

The experimental evaluation was performed using ROPGadget-v3.3 [23], an ROP gadget generator/compiler, as the testing tool. It scans binaries to find useful gadgets for ROP attacks.

Figure 3.8 shows that CFI enforcement is quite effective, resulting in the elimination of the vast majority (93%) of gadgets discovered by this tool. Moreover, there is little diversity in the gadgets found — the tool was able to find only the following gadgets:

Name	Reloc CFI	Strict CFI	Bin CFI	Instr CFI
perlbench	96.62%	96.24%	93.23%	58.65%
bzip2	97.78%	95.56%	93.33%	44.44%
gcc	97.69%	97.69%	91.42%	66.67%
mcf	95.45%	90.91%	90.91%	36.36%
gobmk	98.84%	98.27%	97.69%	70.52%
hmmer	97.00%	96.00%	96.00%	58.00%
sjeng	92.75%	92.75%	91.30%	47.83%
libquantum	93.18%	90.91%	86.36%	40.91%
h264ref	98.26%	97.39%	96.52%	60.87%
omnetpp	97.12%	97.12%	93.42%	74.07%
astar	95.35%	93.02%	93.02%	46.51%
milc	95.77%	94.37%	90.14%	57.75%
namd	94.87%	92.31%	92.31%	53.85%
soplex	94.64%	93.75%	93.75%	54.46%
povray	96.75%	96.75%	95.45%	61.69%
lbm	94.12%	88.24%	88.24%	23.53%
sphinx3	95.00%	93.75%	92.50%	52.50%
average	95.95%	94.41%	92.68%	53.45%

FIGURE 3.8: Gadget elimination in different CFI implementation

- **mov constant, %eax; ret** (32.26%)
- **add offset, %esp; pop %ebx; ret** (27.42%)
- **add offset, %esp; ret** (19.35%)
- **mov (%esp), %ebx; ret** (14.52%)
- **xor %eax, %eax; ret** (5.65%)
- **pop %edx; pop %ecx; pop %ebx; ret** (0.81%)

Among other missing features, note the complete lack of useful arithmetic operations in the identified gadgets. As a result, the tool is unable to build even a single exploit using these gadgets

3.5.3 Performance evaluation on baseline system

We evaluate the baseline performance of PSI compared with two DBI tools: DynamoRIO and Pin. We evaluate runtime performance of the SPEC 2006 benchmarks, including CPU, memory and space overhead. In addition, we also compare micro-benchmark result using `lmbench`. Finally, we evaluate the runtime performance using several benchmarks of well used programs.

Our testbed consists of an Intel core-i5 2410m CPU (Sandybridge) with 4GB memory, running Ubuntu 11.10 (32-bit version), with glibc version 2.13.

3.5.3.1 CPU intensive benchmarks

We used the SPEC 2006 CPU benchmark to evaluate both the runtime overhead and space overhead.

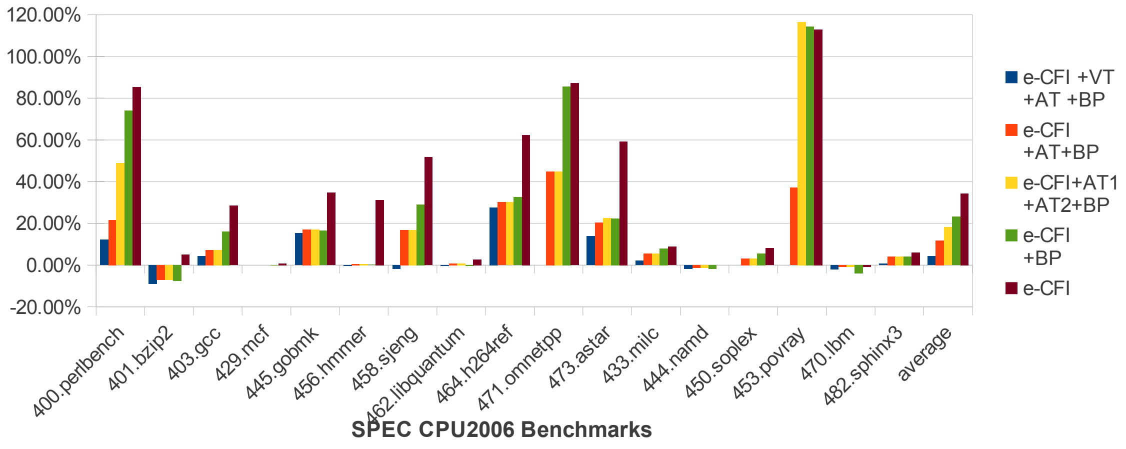

FIGURE 3.9: SPEC CPU2006 Performance with Optimizations

Runtime Overhead In the experiment, we tested different optimizations and their impact. Notations of optimizations in Figure 3.10 follow Section 2.4. The average overhead with all optimizations turned on is 4.29%. In full transparency mode, (ie. with VT turned off), the overhead on average is 11.77%. The overhead goes up to 18.18% if AT.3 is turned off, and 23.2% if all AT options (AT.1, AT.2 and AT.3) are turned off. Finally, the average performance overhead goes to 34.33%, if transparent call and ret (BP) are not used.

In non-transparent mode, C++ programs such as 471.omnetpp, 450.soplex, 453.povray do not work, due to C++ exception handling. Although the issue could be solved by patching DWARF meta data in ELF binaries, it hasn't been implemented yet. Among C++ programs, the worst performance observed is the 453.povray. This is because this program contains a significant number of indirect branches that need translation at runtime. Overall, the best runtime performance on SPEC CPU 2006 benchmarks achieved is shown in Figure 3.10. The average overhead for C programs is 4.29%. Due to C++ exception handling, VT.2 (Section 2.4.3) cannot be applied to C++ programs. As a result, the overhead for C++ programs increases to an average of 8.54%.

Note that this final result may not be accurate on all CPU models. This is because some of the optimizations are CPU dependent. For instance, the effect of branch prediction improvement (BP) is not quite visible on Intel Haswell CPU series. In compare, when testing on AMD Opteron 62xx and Intel core-i5 2400m (Sandybridge), we find that the effect of branch prediction optimization is instead visible. This is mainly because of the difference on the branch prediction ability among different CPU models. Generally, a CPU that has a better branch prediction ability for indirect jump will minimize the effect of BP.

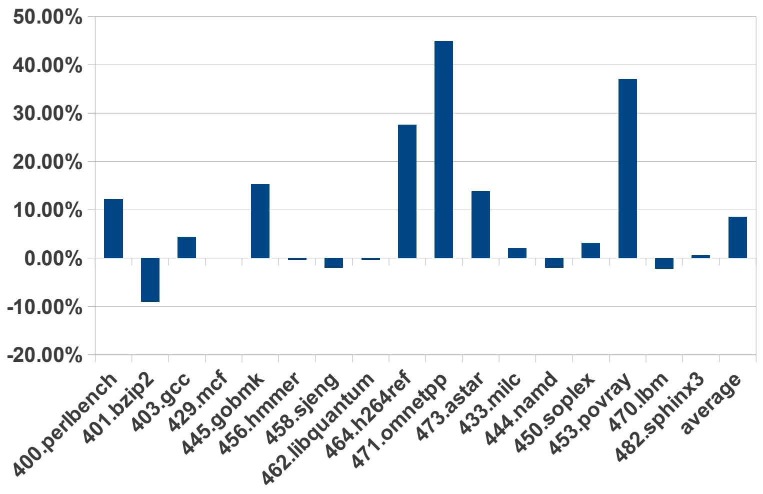

Figure 3.10: SPEC CPU2006 Benchmark Performance

Space and Memory Overhead We also measured the space overhead of DynamoRIO and Pin. We found that their address space overhead is 272% and 72%. This is mainly because the DBI tools need to reserve address space for code cache allocation. The physical memory overhead is 7.5% and 34%, higher than PSI.

Evaluation of System Call Policy On the SPEC 2006 benchmark, the baseline system call policy enforcement introduced an average of 1.6% overhead. This is performed with a policy function registered for each system call, but the function body being empty.

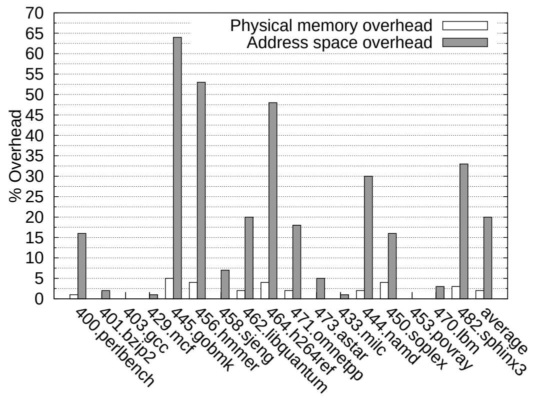

FIGURE 3.11: Space overhead of PSI on SPEC 2006

3.5.3.2 Microbenchmarks

Although DynamoRIO performs well on CPU-intensive SPEC2006, real world programs often exhibit different characteristics. To compare PSI with DynamoRIO and Pin for workloads that are system-call intensive, we used the `lmbench` [17] benchmark. Since `lmbench` (as well as the real-world evaluation in the next section) caused some DBI platforms to slow to the point where experiments took far too long to complete, null instrumentation in these experiments is used to minimize their runtime.

Figure 3.12 shows the `lmbench` performance numbers. Note that the histogram is drawn to logarithmic scale on Y axis. The average system call overhead for PSI is 16.9%, whereas for DynamoRIO it is 312%, and for Pin it is 3300%. On system calls related to communication, PSI achieves almost native performance, whereas DynamoRIO has 36.1% overhead, and Pin has 378%. System calls related to signal handling and process spawning slow down PSI by 43.3% and 79.7% respectively, while the corresponding number is 222% and 948% for DynamoRIO, and 104x and 198x for Pin.

To summarize, the average overhead across the tests shown in Figure 3.12, while counting only one of the `select` operations (for 10 fds), is 33% for PSI (geometric mean: 30%), while for DynamoRIO it is 413% (geometric mean: 309%) and for Pin it is 7873% (geometric mean: 4083%).

DBI platforms are complex, and hence the reasons for their high overheads on lmbench (and the real-world applications discussed in the next section) aren't

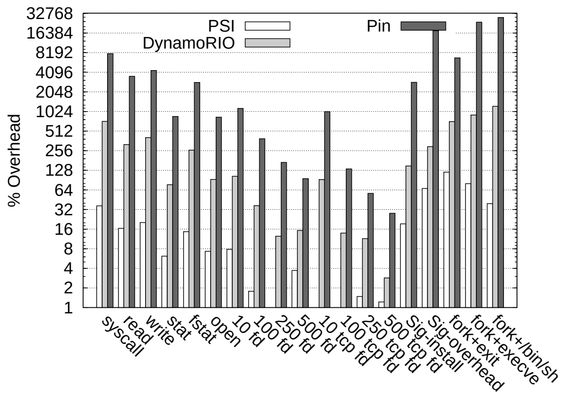

FIGURE 3.12: lmbench microbenchmark result

all obvious. But there are several factors that contribute to the high overhead. First, and most obvious, is that runtime disassembly and instrumentation incurs nontrivial overheads, unless this cost is amortized across many executions of instrumented code. Such amortization occurs on CPU-intensive benchmarks, but the real-world applications discussed in the next section tend to load much more code, and execute it far fewer times, thus causing the overheads of runtime instrumentation to rise significantly. This is one of the main reasons for the high overhead of applications that make frequent calls to `execve`.

A second factor that contributes to the overhead is the increased memory footprint of dynamically instrumented programs. Experiment shows that DynamoRIO can frequently use a code cache that is over a 100MB in size. Moreover, such increased use of data memory can significantly slow down `fork` due to factors such as the increased time for copying page tables.

A third factor relates to threads and locking that is needed to ensure that accesses to data used by the DBI platform (including the code cache) are free of concurrency errors.

3.5.3.3 Commonly used applications

In this section, we performed a comparative evaluation on the performance of PSI with that of Pin and DynamoRIO using a collection of commonly used applications. Once again the null instrumentation is used to minimize the runtime for the experiments.

Program	PSI (%)	Dynamo-RIO (%)	Pin (%)	Description
`coreutils`	97%	1922%	3509%	Coreutils testsuite
`gcc`	63%	1376%	10250%	Compile openssh.
`apt − get update`	2%	326%	411%	Run command 5 times.
`enscript`	211%	5292%	15153%	Convert text and source code files to ps and pdf. [a]
`postmark`	2%	22%	64%	Run benchmark.
`gpg`	24%	382%	5994%	Operate on pdfs with an avg size of 500KB.
`tar`	19%	79%	1107%	Tar /usr/include.
`find`	21%	34%	38%	Find a file in /.
`scp`	-1%	18%	31%	Copy 10 mp3s, with an avg size of 5MB.
`mplayer`	32%	67%	211%	Play 10 mp3s.
`vim`	56%	92%	615%	Search and replace strings in 18MB text file.
`latex`	51%	185%	1806%	Compile tex files with an avg size of 17KB to dvi.
`readelf`	62%	71%	197%	Parse the DWARF sections of `glibc`.
`python`	33%	85%	96%	Run pystone 1.1 benchmark.
Average	**53%**	**887%**	**3421%**	

[a] a bunch of text files are used such that the their file sizes average to 8K. In the experiment, two types of source files are used for the purpose: C programs and Python programs. We averaged C file sizes from Openssl source package and used the same average of 18K for the test purpose. For Python, we averaged sizes of Python scripts found on a typical Ubuntu machine as 8K and used a file in the same size for the test purpose.

FIGURE 3.13: Real World Program Performance

We first measured the performance for two tasks that are commonly undertaken by typical Unix users: compilation of software, and running scripts that invoke other programs. Specifically, we compiled OpenSSH with GNU `make` and `gcc` tool chain, and used the built-in testsuite of `coreutils`.

When testing gcc compilation with PSI, we instrumented all the executables in the gcc toolchain, including `gcc`, `g++`, `cc1`, `cc1plus`, `f951`, `lto`, `ar`, `ranlib`, `as`, `ld.bfd`, and `collect2`. The work also instrumented `make` and all external tools used in makefile such as `echo`, `sed`, `cat`, `perl`, and `gawk`. Note that all libraries used by these programs were transformed too. The overhead incurred by PSI was 63%, while DynamoRIO and Pin incurred overheads of 1376% and 10250% respectively.

In the case of `coreutils`, for PSI, we transformed all `coreutils` binaries as well as other programs used in the coreutils test suite. But due to difficulties in invoking DynamoRIO on each `coreutils` program inside the test script, the experiment used DynamoRIO to run `make` so that it will subsequently instrument all programs invoked from there. DynamoRIO incurred a 19.2x slowdown, as compared to 97% for PSI. When Pin tested , unfortunately, the testsuite did not stop after running for over an hour. We stopped the experiment at this point, and used that figure as the lower bound on runtime of Pin, which worked out to be 3509%.

In addition to the above tests, we measured the overhead for several commonly used programs. The results are shown in Figure 3.13. It is worth mentioning that `apt-get` update invoked several executables including `http`, `gpgv`, `dpkg` and `touch`. All of the executables as well as libraries used were transformed in the tests.

Note that for about half the applications, there is more than 10x difference between PSI and DynamoRIO. The difference drops down to 3x to 5x for about a quarter of the applications, and for the remaining, the overhead difference is within a factor of two. When averaged across all of the applications, PSI's overhead is 53% (45% geometric mean), DynamoRIO's overhead is 887% (322% geometric mean), and Pin's overhead is 3421% (924% geometric mean).

In addition to the runtime overhead on the SPEC 2006 benchmark, we also measured space overhead of the platform. Figure 3.11 describes the space overhead of the platform on the SPEC 2006 benchmark. From the figure, the virtual memory overhead of the platform is 19.79%, while the physical memory overhead is merely 1.68%. In addition to the space overhead for physical and virtual memory, we also measured that the platform increased the on-disk size of the executables and shared libraries by around 139%. This is because PSI instruments a copy of original code, leaving the original code in place. However, this original copy is seldom accessed, which explains why the resident memory overhead is very small, at around 2%.

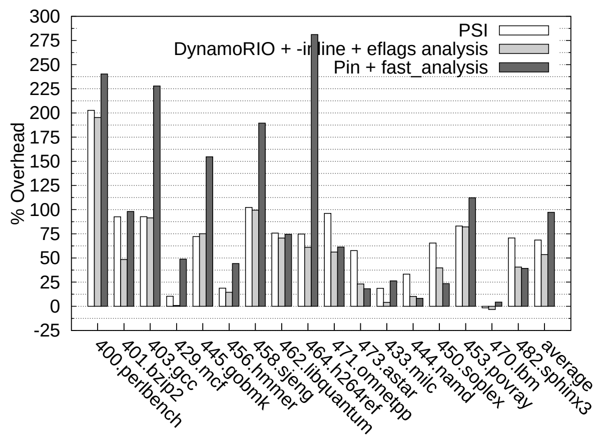

Figure 3.14: Overhead of basic block counting application of PSI, DynamoRIO, and Pin on SPEC 2006.

3.5.4 Performance evaluation on sample instrumentations

This section focuses on comparative evaluation on the performance of PSI and DBI tools running with sample instrumentations. The experiments illustrate the runtime overhead of two instrumentations: basic block counting and shadow stack.

3.5.4.1 Basic-block counting

The instrumentation tool developed incorporates a simple optimization to skip flag saving in some common cases, but lacks a systematic liveness analysis. In spite of lacking this optimization, performance of PSI (average overhead of 69%) is only slightly worse than DynamoRIO (53%), and better than Pin (97%). The result is shown in Figure 3.14. For this experiment, we used the most optimized version of basic-block counting applications distributed with Pin and DynamoRIO platforms.

Although PSI is designed for offline instrumentation, we turned on the on-demand instrumentation feature, emptied the library cache and reran the benchmark. In addition, we added back the time for instrumenting the executables to the totals. In this way, we measured the total runtime that includes instrumentation time. This change causes the overhead to increase by another 3%.

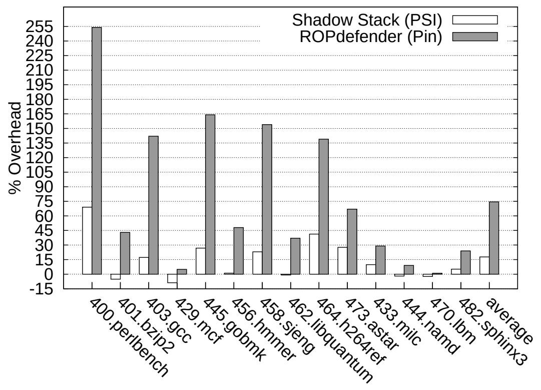

FIGURE 3.15: Overhead of shadow stack implementation using PSI and that of ROPdefender on SPEC 2006

3.5.4.2 Shadow stack

Figure 3.15 depicts the overhead of the shadow stack implementation using the platform in comparison to that of ROPdefender, which is based on Pin. While ROPdefender reports an overhead of 74%, PSI incurs just one-fourth of this overhead (18%).

Prasad et al [115] report lower overheads, as low as a few percent. But as mentioned earlier, their technique does not defend against ROP attacks.

3.6 Summary

In this chapter, we have presented a general static binary instrumentation platform, PSI. We have elaborated its internal design and its API. In addition, we highlighted on-demand instrumentation as an important feature. Then, we have demonstrated that PSI could be used with various instrumentation.

Finally, we have presented an extensive evaluation of our system PSI. We evaluated our static instrumentation system functionality using static checking and dynamic checking. We then presented our security evaluation. And we further compared our system with modern binary instrumentation systems such as DynamoRIO and Pin. We compared with them on both the baseline instrumentation and several instrumentation applications. The results demonstrated that PSI achieves almost the same result on SPEC benchmark and much better performance on commonly used applications.

Chapter 4

Application: Practical Protection From Code Reuse Attacks

Previous chapters are focused on discussion of essential elements of the binary instrumentation platform PSI. Those techniques form the foundation that allows complex security instrumentation and extensions to be realized. We includes two significant applications built with PSI platform in this chapter and the next chapter. In this chapter, we focus our discussion on code reuse attacks.

In this chapter, we propose a practical defense against advanced code reuse attacks by introducing several hardening methods. For instance, we tighten PSI policy by eliminating targets used by jump tables from the allowed set of return. In addition, we use *code pointer remapping* (details in Section 4.4) to defeat attacker that leverage their prior knowledge of code layout to construct gadgets. In addition, we achieves the "executable-but-not-readable" security property without additional support from OS or hardware. This is achieved using a novel technique called *code space isolation* (details in Section 4.5). We implement these features in a system called SECRET. SECRET is built on top of PSI and offers a strong protection against ROP attacks on COTS binaries.

4.1 Motivation

The deployment of non-executable page protections in recent operating systems prompted a shift to code reuse attacks [65, 85, 126]. After hijacking the control flow, execution can be diverted to code that already exists in the address space of the vulnerable process. Return-oriented programming (ROP) [126] has become the de facto code reuse technique, as the stitching of short instruction sequences (called "gadgets") allows for increased flexibility in achieving arbitrary code execution.

Recent research illustrates an intensive arm race on code reuse attacks [46, 48, 55, 56, 62, 68, 73, 74, 120, 126, 130] and defenses [28, 36, 37, 61, 63, 71, 86, 97, 105, 110, 136]. Control flow integrity is regarded as a principled approach against control flow hijacking attacks including code reuse. However, recent research has shown that practical coarse grained CFI cannot defeat them [55, 62, 72, 73]. This is because ROP attackers could still use allowed targets to construct gadgets and successfully launch ROP attacks. Researchers have proposed their work to improve the strength of CFI policy [94, 105, 106]. However, it is known that improving the policy is at the cost of compatibility loss. For instance, MCFI [105] tightens CFI policies using type based checking and suffers from false positives [136].

Despite the fact that there are still gadgets when CFI is applied, to find them, attackers usually need knowledge of location and structure of existing code. Code randomization [44, 45, 64, 75, 83, 109, 140] is an orthogonal research direction that helps randomizing the load address of binaries and the structure of the code itself, so that even if the location of some piece of code is known, the rest of the code functionality remains unknown. However, these approaches in principle suffer from memory disclosure bugs. Attackers could leverage this type of vulnerabilities to adjust gadget addresses in a ROP payload to achieve reliable execution [27, 43, 88, 125]. Going one step further, malicious script code can leverage a memory leak to dynamically scan the code segments of a process, pinpoint valid gadgets allowed by CFI, and synthesize them into a functional ROP payload. Such "just-in-time" ROP (JIT-ROP) attacks [130] can also be used to bypass fine grained code diversification protections. Therefore, code randomization alone cannot help strengthening CFI policy. On the other hand, recent research illustrates that instead of relying on memory disclosure, attackers could use the crash-restart property of some server programs to enumerate all possible targets [46].

In sum, the key point in advanced code reuse attack is the way to construct a gadget chain. Attacker may choose various methods to construct such a chain: (a) use prior knowledge on the binary layout; (b) blindly enumerate and try using existing gadgets; (c) dynamically read code pages, or even (d) harvesting code pointers.

In this dissertation, we propose an alternative approach on top of BinCFI to prevent modern code reused attacks that is guided by memory disclosure or blinding probe capability. Our system is called SECRET which improves our baseline system in three aspects: (a) tightening baseline CFI policy without sacrificing compatibility, (b) introducing code pointer diversity and (c) removing code reading capability from attackers.

Our approach tightens the BinCFI policy by eliminating jump table targets from the valid target set for all return instructions. In addition, our approach uses *code pointer remapping (CPR)* as a practical defense against conditions that may allow for inferring the functionality or probabilistically pinpointing the location of gadgets, e.g., through leaked return addresses from the stack. In other words, code pointer remapping aims to mitigate code pointer harvesting attacks. This is achieved by mapping the values of code pointers into another randomly selected address space that only instrumented code can translate. Code pointers such as return addresses are randomly mapped into this address space, making inference of valid code locations from leaked code pointer value challenging. In addition, the projected address space is much larger than the original code segment, thus making the number for tries required by blind ROP several orders of magnitude higher.

In addition to code pointer remapping, our approach eliminates code reading capability from ROP attackers. This is achieved by *Code space isolation (CSI)*. It prevents code reading accesses to both original code and instrumented code. Accesses to original code are prevented by eliminating machine code in original code segment. The scope of original code is identified using our static analysis. On the other hand, preventing read accesses to instrumented code is achieved using segmentation in x86 systems. In x86-64 systems, it is achieved using probabilistic protection through randomized placement in the vast address space.

SECRET is built on top of PSI. To make deployment easier, SECRET does not require modification on original binary. Instead, instrumented code is independently loaded at runtime and the original binary remains the same on disk until load time, when it is changed by our dynamic loader. The results of experimental evaluation with the SPEC benchmarks and real-world applications demonstrate that this protection introduces only up to less than 2% additional runtime overhead on top of PSI, bringing the overall average overhead to 14.5%.

4.2 Threat model

We consider a powerful remote attacker who has the ability to execute arbitrary script code that can exercise memory disclosure or corruption vulnerabilities in a victim application. We assume that the attacker has no physical access or native code execution capability on the victim system. We also assume that victim system has deployed ASLR and DEP and thus direct code injecting does not work, and hence the attacker's goal is to carry out a ROP attack.

We further assume that using the memory disclosure capability, the attacker can systematically de-randomize the locations of data and code that is accessible by following pointers stored on the stack, the heap, and global data in ELF images.

The ROP attacker can choose several ways to succeed. The simplest one is to use memory disclosure to harvest code pointers and use existing leaked code pointers to infer the layout of binary and finally construct ROP attack at runtime. This attack is straightforward but may not be robust, since binary code layout may vary across different distributions. Alternatively, the attacker can also launch JIT ROP attack which constructs ROP payload just in time. By following direct branches or other leaked pointers (e.g., from `vtable` pointers to virtual function pointers and finally to code pages), she could harvest sufficient code pages to gather ROP payload and launch attacks.

In typical ROP scenarios, victim crashes (caused by invalid memory accesses) lead to attack failure. We consider a more powerful threat model that attacker could repeatedly cause crashes and regain control, similar to blind ROP attack [46].

Finally, we assume that the attacker is aware that a coarse-grained CFI has been applied to all code modules in the target system.

4.3 System design overview

In this section, we demonstrate the design of SECRET against code reuse attacks. We have developed our approach to disrupt the process of constructing gadget chain. Specifically, we have developed the following techniques:

- *Code pointer remapping (CPR):* We develop CPR that replaces the code pointers including return addresses, exception handlers, exported functions with encrypted values that scatter in a very large address space. By doing so, attackers won't be able to infer valid indirect branch locations from leaked code pointers in the stack or heap. In addition, encrypted code pointers live in a large per-module address space so that blind ROP attacks will suffer from significant number of tries. The detailed design is in Section 4.4.

- *Code space isolation (CSI):* We eliminate attackers' capability of code reading by using code space isolation. In particular, we first use static analysis to figure out original code and wipe it out. In addition, we present techniques to isolate and protect our instrumented code from being revealed. Read access to the instrumented code version is prevented on x86-32 using segmentation features. On x86-64, we exploit the available address space entropy to achieve probabilistic protection. The design of CSI is detailed in Section 4.5. In addition to protecting instrumented code, CSI ensures that instrumented code cannot be discovered by following pointers on the stack, static, or heap memory.

4.4 Code pointer remapping

In order to launch code reuse attacks, ROP attackers need to find useful gadgets to construct a ROP chain. However, in the context of a coarse grained CFI, only valid indirect targets can be used for gadgets. This does not stop them because if attacker has prior knowledge of original binary, he could use gadgets that live in valid targets permitted by BinCFI. For non position independent executables, attackers have no problem figuring out locations of those gadgets using prior knowledge. To gather gadgets in position independent code, all they need to do is to bypass ASLR, i.e., once any code pointer is leaked, attackers may use their knowledge to infer all other code pointers that reside the same module.

In this section, we focus on solving this important issue and convey a defense against ROP attacks that bypass ASLR. Our approach is called code pointer remapping (CPR). The key point in CPR is that it aims to prevent attackers' prior knowledge on original binary from being useful. This is achieved by the following countermeasure steps:

- *Preventing usage of invalid pointers:* PSI already enforces a coarse-grained CFI policy that prevents control transfers to most invalid targets. The vast majority of the allowed targets are the instructions following a call instruction. The remaining are code pointers discovered by a conservative static analysis. Control transfers to any other location (98%) are blocked.

- *Tightening BinCFI policy:* We use static analysis to tighten BinCFI so that its strength can be stronger without sacrificing compatibility. On implementation, we could use multiple address translation tables for different indirect branches., For instance, indirect jumps using only jump tables could check a separate table, so that those code pointers in jump tables can be eliminated from the address translation table for return instruction.

- *Remapping code pointers:* Since instructions after call sites constitute the majority of allowed targets, they are the most attractive targets for attackers. We present a technique that replaces return addresses in each module into values in a new code address space (address range). Each code pointer is encrypted using a strong hash function such as SHA-2. The encrypted value in each module is then mapped into one address space so that attackers cannot infer additional pointers from leaked ones.[1] In addition, the new address space is randomly selected and reserved by ld.so and its size is several orders of magnitude larger than original code segment. Thus, exhaustively probing memory for callsite gadgets won't work.

Since the first step has already been covered by the underlying system. We focus our discussion on the rest of them.

4.4.1 Tightening baseline policy

We discover that part of BinCFI policy is overly permissive. In particular, return instructions does not need to use jump table targets. However, elimination of these addresses from address translation table cause some indirect jumps that use jump tables not working, since they share the same table. To overcome this, we change the transformation of those indirect jumps. In particular, we transform code pointers used by jump tables and put them into a new table along with instrumented code. In addition, we change indirect jumps that look up old jump

[1] By masking the encrypted value, we could control the range of encrypted pointer values for each module. In some scenarios, such as handling exceptions, this range will have to include several smaller ranges, each of which corresponds to one region specified by DWARF information and all return addresses in the same range will have to preserve order. This can be easily achieved by tweaking the mask values.

tables to ensure that they check the corresponding new tables in instrumented code. By doing so, jump tables will enjoy better performance since no translation is needed. In addition, jump table targets are eliminated from the address translation table to improve the strength of CFI policy.

4.4.2 Remapping code pointers

To prevent attacker from using leaked code pointers to infer the code layout, SECRET remaps original code pointers of ELF binaries including return addresses on the stack as well as other types of pointers such as exported functions, and discovered jump tables, as this provides the following benefits:

- Remapping code pointers can prevent attackers from being able to easily target the most powerful class of gadgets left on systems that enforce coarse-grained CFI. (On such systems, code pointers such as return address can target far more gadgets than other types of code pointers. Moreover, gadgets reachable using returns can be assembled into non-trivial ROP payloads.)
- Attackers may be able to launch an ROP attack if they could systematically enumerate the space of valid code pointer values, and then probe them all. By remapping the RA as well as other code pointers, we force the attacker towards exhaustive enumeration: return addresses and other code pointers that are next to each other in the original code address space are now scattered across a much larger and randomized address range, forcing attackers to brute-force the entire address range rather than being able to use an intelligent search that probes only a small fraction of the address space.

Remapping code pointers in the following means that we use a strong hash method such as SHA-2 to encrypt the target code pointer and use a mask to map them into a certain address range.

To figure out a new code space for each module, we leverage the dynamic loader to reserve an address range. The base address of this address range is decided at runtime, but on instrumentation time, there is always a predetermined base which is the next page address after the current code segment. Since code pointers are modified at instrumentation time, these pointers are mapped into this predetermined address range and then they are rebased further at runtime through relocations. Relocations are contained in a file loaded along with instrumented code. For simplicity, each module owns an address space of 4 Gigabytes.

Within the address range, we generate a random number for each code pointer to be modified. Since our address space is 4 Gigabytes on default, all we need is a set of 32-bit random numbers.[2] Instrumented programs will have to use these random number instead of original code pointers. LTT is also modified to make sure those randomized addresses will be found in the key set at runtime. All of these randomized code pointers will be translated at runtime into addresses within instrumented code. Attackers won't be able to reverse engineer the address translation since the location of LTT and instrumented code are hided within the large address space.

Remapping return addresses Changing return address (RA) values has two potential implications. First, existing code may rely on RA values, e.g., exception handling code. The experiments have shown that the use of RAs is limited to the following cases on GNU/Linux:

- *C++ exception handling,* where the return address is used to identify whether the caller of the current function has a handler for the current exception,
- *Location Checking,* where the return address is used to check where the caller comes from. This type of checks happen in dynamic loader.
- *PIC code data access,* there are two cases: (a) jump tables, where the return address is popped off the stack and used to compute the base address of a jump table, and (b) static data accesses where the return address is popped off the stack and an offset added to find the base address for static data access.

For cases where return addresses are used for C++ exception handling, we update the `DWARF` metadata information. This is to ensure that the stack unwinding mechanism can work correctly with randomized return addresses. In particular, we update the `DWARF` for each function by changing the function boundaries. The randomized function boundaries are equally distributed in the large random region for the whole module. Since C++ exception handling only checks return address with its own function boundaries, the order of function ranges does not have to

[2] Note that this address space can be configured to increase randomness

be preserved. To make sure each return address is mapped to its corresponding function, we tweak the mast value to achieve that. This sacrifices the randomness of return addresses from arbitrary value within a whole module (4Gigabtes) to those within the range of its function (4 Gigabytes divided by number of functions in this module). We argue that this can be solved by enlarging the address range of the whole module.

The second case that we observe occurs in the dynamic loader, where some internal functions check the location of caller. In particular, they require that callers could only come from `libc.so` or `libpthread.so`.[3] In particular, the code will check against the loader's internal data structure `link_map`, which contains the information about all modules. In order to cope with this, we change the `link`$_{\mathtt{m}}$`ap` data structures. In particular, the base address to that of the randomized address space. By doing so, the remapped return addresses could be correctly identified. In addition, to make sure all metadata could be accessed, we also change related related fields in `link_map` such as offsets to metadata segments of the module accordingly. This is to ensure that our modifications is transparent in front of accesses towards ELF metadata sections

For the other two cases, we rely on a static analysis to detect that the RA is being used as a data pointer, and avoid remapping the RA in those cases. Moreover, it is possible to recognize that those addresses will not be used for return (as they are popped off the stack), and hence avoid including them in the list of valid targets for return instructions.

Another challenge in remapping RA is that of handling special cases of instructions: instructions that use an RA that wasn't pushed by a call instruction. A typical example is that of explicitly storing a code address at the top of the stack, and returning to it. The return address in such case will not be remapped, which can cause incompatibility, since the instrumentation now requires remapped RAs. We point out that any such code address that is explicitly stored on the stack will be identified as a code pointer by the static analysis used in PSI. Moreover, we point out that the coarse-grained CFI policy in PSI permits return instructions to return to locations identified by the code pointer analysis. In terms of implementation, this policy is implemented using two address translation tables: one for translating return addresses, and another for translating other code pointers.

[3]Somehow, this is regarded as a security feature, which is obviously useless. Attackers could bypass the checks by getting into the middle of those functions directly, or carefully crafting the payload to bypass the checks.

(The LTT components of these two tables, called RA LTT and FP LTT, are illustrated in Figure 4.1.) The approach is to make the return address translation operate on remapped RAs, while letting the code pointer translation operate on non-remapped pointer values. This approach provides an elegant solution to the compatibility problem posed by special cases of return instructions.

Remapping exported functions Remapping exported functions is necessary, since these pointers will be propagated by the dynamic loader into GOT tables in dependent code module. Attacker may use this information to infer other code module base addresses. Remapping these code pointers can be done by updating the dynamic symbol table in each ELF image at runtime.

Handling function pointers Note that function pointers remain unchanged in the ELF binary file. This is because our technique targets COTS binaries that do not contain additional information such as relocation information or static symbol table. Without the additional information, it is challenging to identify whether a constant value is a code pointer or an integer. Therefore, we leave them unchanged. However, an on-going work is being developed to identifying vtables and transformed code pointers in vtable. This analysis has also been attempted by previous work [114].

4.5 Code space isolation

Code pointer remapping prevents code pointer harvesting attacks, i.e., attacks that infer other pointers from leaked ones. However, bypassing code pointer remapping is easy, since attackers could simply read original code as well as artifacts specific to PSI such as instrumented code and LTT to figure out valid indirect branch targets. For instance, discovering original code allows attackers to find all useful gadgets and discovering LTT allows attackers to find out all indirect targets permitted by BinCFI. Thus any effect of code pointer remapping will be useless.

To make code pointer remapping robust against memory leakage, the first thing we can do is to eliminate original code, since it is not useful anymore in PSI. In addition, we need to ensure that those encrypted pointers remain secret. This requires hiding both the LTT and encrypted return addresses in the callsites. An easy way to achieve the latter is to use a table containing all encrypted return addresses and ensure that instrumented code for call uses its index instead of encrypted value to push return addresses. (e.g., `push $encrypt_ptr` should be rewritten to `push array[index_i]`). However, the first one requires randomizing LTT without direct memory access. Hiding LTT in this way unfortunately puts further constrains on our address translation trampolines. Since the address translation routine needs several indirections to get the base address of LTT, the extra runtime overhead will be significant.

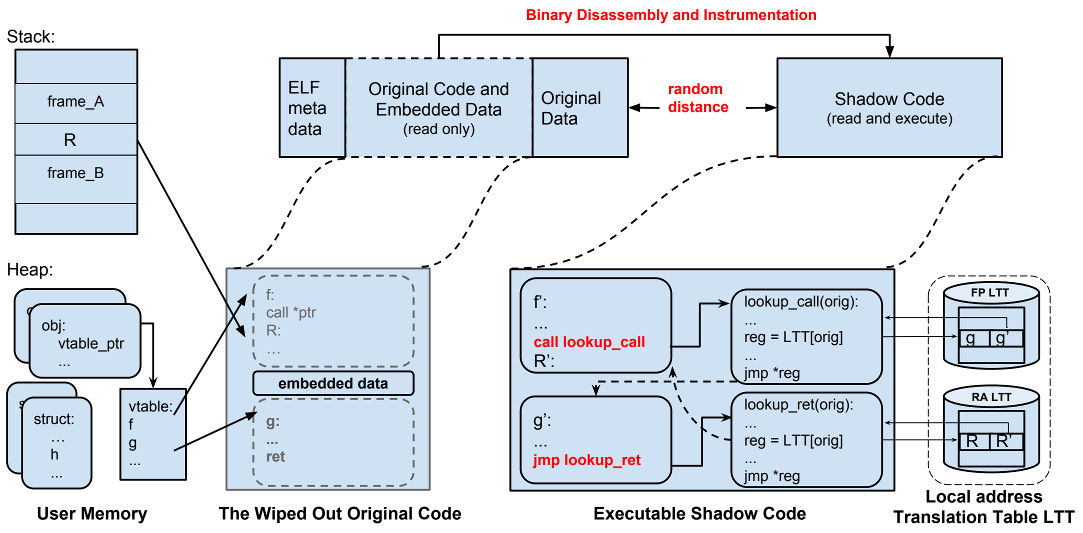
Binary Disassembly and Instrumentation
Stack:
frame_A
R
frame_B
ELF meta data
Original Code and Embedded Data (read only)
Original Data
random distance
Shadow Code (read and execute)
Heap:
obj: vtable_ptr ...
struct: ... h ...
vtable: f g ...
f: call *ptr R: ...
embedded data
g: ... ret
f': ... call lookup_call R':
g': ... jmp lookup_ret
lookup_call(orig): ... reg = LTT[orig] ... jmp *reg
lookup_ret(orig): ... reg = LTT[orig] ... jmp *reg
FP LTT
g g'
RA LTT
R R'
User Memory
The Wiped Out Original Code
Executable Shadow Code
Local address Translation Table LTT

FIGURE 4.1: Architecture of code space isolation. Memory accesses towards address translation table are performed through our private TLS to avoid memory leak.

To protect code pointer remapping from memory disclosure with a good trade-off between performance and effectiveness, we propose code space isolation (CSI). Figure 4.1 illustrates the overall approach for protecting the original and the instrumented code, which relies on the following techniques. First, we develop a static analysis pass to identify data embedded within the original code, and zero out the remaining code bytes. Note that this approach is able to do this at a fine granularity compared to some of the previous approaches, such as XnR [36] and HideM [71], which offer protection only at the granularity of memory pages. For instance, with these approaches, if there are just a few bytes of data in every 4K of code, they can cause all of the code to remain readable. In contrast, with our approach, will all but few data bytes are erased, thus allowing attackers to read just these few bytes.

After erasing code bytes in the original data, we develop techniques to separate instrumented code as well as LTT into a memory region that is unrelated to that of the original code.

4.5.1 Static analysis for identifying data

We have developed a static analysis to identify data embedded within code regions. The goal of this analysis is to be conservative: when in doubt, bytes should be marked as data and preserved, rather than being erased.

There are two types of embedded data: (a) data in the middle of a function and (b) data between functions. The first type of data is usually jump table data. We reuse the underlying PSI jump table discovery code to discover the CFG of a function, and mark the gap inside the function as data. For the second type of data, we leverage the information in two sections in COTS ELF binaries: `.eh_frame` and `.eh_frame_header`. These two sections are generated in the `DWARF` format used for C++ exception handling at runtime. These sections tell the C++ runtime how to unwind function frames. Basically, the information is per function based. They include information such as function boundaries, the position of saved frame pointer in a stack frame (or stack height if frame pointer is not available), and the positions of callee-saved registers saved in the frame.

We find that even non-C++ binaries contain these two sections. This is because C++ code may call non-C++ code and vice-versa. In order to properly handle exceptions, all function frames between the exception thrower and catcher must be available. This is the main reason why non-C++ code also includes frame information by default. Although `DWARF` information may be missing in some cases, in the experiments, we find that it is available on COTS Linux binaries. It is included even in low level code such as `glibc`, including even the hand-written assembly code contained therein.

The disassembler implemented is based on the original design in Section 2.1, but we add some tweaks that leverage the DWARF information to improve disassembly result. In particular, we first leverage the `DWARF` information to accumulate all the available function boundaries. Then, we start disassembling the code from those known code locations and follow direct control transfers to discover code that may live in the gap between two known functions.

Once this process is over, we consider the rest of the original code as embedded data.

4.5.2 Separating instrumented code from original code

Since static binary instrumentation approach updates binary files and generates a new code segment, it is natural to extend the existing binary by putting the instrumented code right after the original code segment. However, this is not good for protecting the instrumented code regardless of the method used: efficient protection mechanisms may require protected regions to be located within specific ranges of addresses, or at random locations. We therefore redesigned the format of instrumented binaries to decouple instrumented code as well as the address translation tables from the original code.

To protect instrumented code in x86-32 architectures, we use segmentation to ensure that instrumented code is not accessible by the program. Note that the technique is to put the instrumented code as well as its LTT outside of a memory sandbox enforced by segmentation. The details of this technique has been discussed in Section 5.3.6. We omit repetitive discussion here.[4]

On architectures such as x86-64 where hardware segmentation is missing, software fault isolation (SFI) [138] can be used to protect the instrumented code, but the associated overheads can be significant. Moreover, instrumenting all memory accesses can be an engineering challenge due to the complexity of the x86 instruction set. We therefore take the alternative of base address randomization to protect the instrumented code. There are two key benefits in using randomization for this purpose:

- *High randomization entropy:* x86-64 architecture supports a 48-bit address space, of which 47 bits are usable in user mode. The larger address space increases the potential entropy of ASLR that operating systems could achieve. For instance, from Windows 7 to Windows 8, the entropy of DLL images increases from 8 to 19 and the entropy of executables goes from 8 to 17 [79]. Linux has also improved the entropy of ASLR for mmap(2) from 8 bits in 32-bit systems to 28 bits in 64bit systems [93, 144].

[4]Note that since LTTs are outside of sandbox, memory accesses towards LTTs are required to bypass memory sandbox, this is achieved by using our private segment register

- *Protection from indirect memory disclosure:* Even if code base is randomized, an attacker that can read data memory such as heap or stack may gather harvest code pointers. These pointers still allow the attacker to reach code pages by dereferencing those pointers. We eliminate this attack avenue since code pointers such as return addresses and exported functions are protected by CSI. The rest of the code pointers, including function pointers , only target the wiped out original code segment. Thus, instrumented code is never targeted by any part of the memory, except in our own data structures that are maintained as secret.

4.5.2.1 Protecting instrumented code

In our implementation, instrumented code of each module is located in random distance from its original code. The random number is independently generated for each module. Note that code allocation is performed in loader instead of libc.so, this dedicated code allocation site gives us an option to control the allocation function and ensure a higher entropy of ASLR than OS without affecting allocation of data or other objects.

Despite the fact that most of the instrumented code is position independent, changing the base location of instrumented code requires adjustment on its data references towards the original data section and references to original code even (in case of data embedded in the code). Data references are realized in x86 with either a position independent code (PIC) sequence (get_pc_thunk) or with an instruction that uses %rip with a constant offset. For data accesses through PIC code pattern, we solve it by identifying those patterns and make sure return addresses remain unchanged (pointing to original code). For data references using %rip in none position independent executables, a straightforward idea is to change the usage of %rip into absolute addresses, since absolute address is known at instrumenation time. For %rip usage in shared libraries and position independent executables (PIE), a natural solution is to adjust the offset value in those instructions using %rip. Unfortunately, according to Intel manual on x86 ISA, the offset value of %rip has only 4 bytes in length, thus adjusting offset value will constrain the range of instrumented code to 4 gigabytes around the original data segment. This solution brings about least code instrumentation but downgrades the entropy of instrumented code location. An alternative solution is to use a register to replace the usage of %rip. The content of this register is the base address of that module. By doing that, a data reference of %rip plus a constant A, can be statically replaced by the register with constant B, where B equals to $\%rip - base + A$.[5] Note that the dedicated register is reloaded each time it enters a new module. An instruction that uses the dedicated register need to be replaced by either an instruction using another register or an instruction sequence with store and reload dedicated register. Since PIC code pattern requires patching the instrumented code, our loader does that before loading it into memory.

4.5.2.2 Protecting global address translation table

Note that in addition to protecting instrumented code, the locations of GTT and LTTs need to be randomized as well. Since LTTs occur together with instrumented code, the entropy in their addresses is the same as that of instrumented code. GTT, on the other hand, is implemented using an one-level array, in which each entry represent one page in the whole 47-bit address. Since the entry size is 16 Bytes, the total size of GTT will be half of one TB. Despite the fact that GTT is still a small data chunk compared with the large address space in x86-64 architecture, the entropy goes down to 9 bits and suffers from memory disclosure, as shown in recent research [68]. Using different table structures such as two-level lookup table or hash table will increase the entropy but definitely will cause performance issues. To avoid extra performance overhead and maintain a high entropy of our secret, we choose an alternative approach, i.e., exposing GTT to both application code and attackers. The point of our design is that GTT does not contain any secret. Note that GTT is a hash table that is looked up by a target page address. Its output is a trampoline address for the target module. Although this trampoline helps completing the address translation for inter-module control flow, it leaks the location of instrumented code of target module. This is because the trampoline code is appended after instrumented code. Decoupling this trampoline from an instrumented code module would be impractical since all indirect branches will have to check against it. To solve this issue, we choose to use index to represent the address of trampoline in the GTT. This index is used for checking against an internal data structure called *trampoline table*. This table contains the trampoline addresses of all code modules loaded and its base address. The index is incremented each time a new module is loaded. Given the fact that each module contains two trampolines, according to BinCFI design, each of which

[5]Note that $\%rip - base$ is always a constant

requires 8 bytes, we further assume that an application will not load more than 500 libraries. Under this assumption, two consecutive pages will be sufficient to the trampoline table and the size of the table could be further configured. Base address of trampoline table is kept in our private TLS whose base is stored in kernel. By doing that, both the GTT table and its lookup code are safe to be allocated at fixed location which is visible and accessible to application code and ROP attackers.

4.5.2.3 Protecting internal data structures

The internal data structures that should remain secret are our private TLS and the trampoline table. Our private TLS is initialized in the loader at program launch time and each time a new thread is created. It uses the TLS segment register that is not used by glibc, which is %gs in x86-64 in Linux. The purpose of this TLS is to store pointers that benefit instrumented code execution. Therefore, its content has to remain secret.

4.5.3 Randomization

As mentioned, the trampoline table contains critical information and thus should be protected. We ensure that this table has a randomized base address and is read-only in the program execution, except in a small window when the program is initializing a library. To summarize, the total number of secret are the following data elements:

- *Instrumented code and LTT:* Instrumented code needs to remain as secret to prevent disclosure of encrypted code pointers.
- *Trampoline table:* This per-process table contains all the address translation trampolines and base address of instrumented code.
- *Private TLS:* Private TLS contains pointers to trampoline table and thus needs to be protected.

We now describe our critical data randomization implementation. Note that Linux implements its own ASLR, which offers 28 bits of entropy [93, 144]. However, bugs [91, 92] and weakness of its implementation plague around the production systems. Even to date, current 64bit Linux system still has a vulnerability in ASLR that PIE is always allocated at the same distance from its loader and libc [93].

For this reason, we decide to develop our own randomization module to allocate instrumented code, the trampoline table and our own dedicated TLS. For simplicity of implementation, we use the assembly instruction `rdrand` available in Intel CPU Ivy bridge models to generate random numbers. `rdrand` is compliant with security and cryptographic standards such as NIST SP 800-90A [40]. This instructions could generate 64bit random number each time it is invoked. Since our code and data have to be page aligned, the number of randomized bits needed in 47 bit of user address space is 35, which we use.[6] By doing so, the instrumented code as well as our critical data structures could enjoy the full user address space.

In our implementation, we add our own code logic in the dynamic loader just after the library loading code. So, whenever an original code module is loaded, it immediately triggers our own code to load its corresponding instrumented code at random base location. Note that by doing so, we do not change the entropy of original code. This is reasonable since increasing the entropy of original code does not help because of the leaked pointers.

4.6 Implementation

Since the base platform PSI works only on x86-32 Linux, we also implemented SECRET on the same platform. It implemented both segmentation-based and randomization-based protection for instrumented code and other memory regions that critically rely on protection. Most of the main steps and issues in the implementation have been discussed in the previous section, so in this section we focus on some lower level issues that are nevertheless critical for ensuring overall security and correctness of SECRET.

Protecting the Dynamic Loader Shadow code loading is performed by a custom version of the dynamic loader (ld.so), which is a shared library responsible for loading dependent library modules. However, since the loader is responsible for loading the instrumented code of other modules, a challenge that must be

[6]If target CPU does not support this instruction, we could use `/dev/random` to get random numbers.

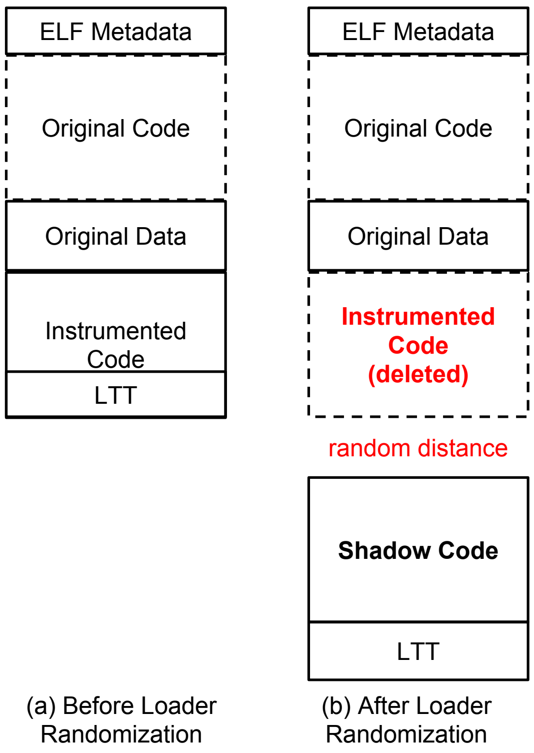

FIGURE 4.2: Shadow Code randomization for the dynamic loader.

addressed is how instrumented code randomization can be applied to the loader itself.

SECRET addresses this issue with the process illustrated in Figure 4.2. When the program starts, the loader runs first, and at this point, its instrumented code and LTT are by default appended after its original data segment, as up to that point the code is just a traditional PSI transformed binary. Immediately after the loader is initialized, it begins to randomize itself. First, a pre-generated instrumented code image is loaded from disk and mapped into a randomly selected address picked by the OS.[7] Similarly, a new instrumented code as well as its LTT data file is loaded afterwards, and then the loader begins to update the GTT. In particular, all entries corresponding to the original code of the loader are changed to point to the new instrumented code. After this step is finished, the loader continues its execution using the original instrumented code until the next indirect control flow is encountered. Once an indirect branch is encountered, the loader "enters" the randomized instrumented code and runs with that code. The final step is to

[7]In case when the entropy provided by the OS is insufficient, this design allows for additional randomization using a custom allocator.

unmap all the old instrumented code and LTT. This is safe since the program is still in its startup phase, and no other threads will be executing the loader's code.

Protecting the vDSO The virtual dynamic shared object (vDSO) is a small shared library automatically mapped by the kernel into all user-space processes. As such, it does not have a file on disk. If left unprotected, however, several critical functions in vDSO could be abused for the construction of ROP code.

SECRET instruments all dependent libraries, including vDSO. This is achieved by dumping its memory image and generating the corresponding instrumented code offline. At runtime, SECRET loads the instrumented code for vDSO at a random location, following a process similar to the one for the dynamic loader, described earlier. This ensures that forward edge control flow transfers will never target the original code of vDSO, and will always be redirected to its instrumented code. The instrumented code of vDSO also benefits from the underlying PSI protection. Moreover, additional security policies could be added to prevent attacks that abuse `sigreturn` in vDSO.

Another complication for vDSO's instrumented code is that vDSO contains a fast system call invocation function. Due to the kernel design, when a system call finishes, the control flow always goes back to the original vDSO code even if the call site was in its instrumented code. If not handled properly, this could lead to the execution of an unprotected return instruction. SECRET deals with this issue by dynamically patching this function and making sure it complies with the underlying CFI policy.

Signal Delivery Signal delivery causes the program context (a signal frame) to be dumped in to the user stack. Since the program is executing in its instrumented code, any signal delivery will cause information leakage that will expose the location of the instrumented code. SECRET avoids this issue by transparently intercepting the `sigalt` function and registering its own signal stack. All signal frames are initially placed in a randomized signal stack whose location is never leaked. Then, the transformed signal frame is copied into the user stack (or user-assigned signal frame). The transformation changes the PC address inside the signal frame into the original code address before copying it to the user stack.

4.7 Evaluation

We perform an extensive evaluation on the system using the SPEC benchmarks and real world applications, including GUI applications such as Open Office. All experiments were performed on a 32-bit Ubuntu 12.04 with a 2-core Core i5 CPU and 4 GB RAM.

4.7.1 Code pointer remapping

Figure 4.3 shows the fraction of code pointers that have been remapped when code pointer remapping is applied. Our analysis illustrates that the majority of code pointers are return addresses. To our surprise, our experiments find that almost half of the code pointers in libc.so are used for jump tables. For this library, we almost remapped all the code pointers. Remapped pointers are hided inside instrumented code which is not accessible by attackers.

4.7.2 Static analysis

We evaluate our static analysis technique on code identification. As described, the `.eh_frame` section provides information on how to unwind stack frames. The covered region is consisted of a list of debugging unit, each of which is usually a function or code snippet. The Frame Description Entry (FDE) structure includes the range of the code in each case. Figure 4.4 shows the exception handling information coverage for a set of SPEC binaries and Linux libraries. We summed up the ranges of all entries and showed them in the 2nd column how much code was covered by the `DWARF` information. On average, 97.17% of the code segments is covered, which means that function boundaries are available in almost all of these binaries.

With those boundaries as starting points, SECRET's static analysis pass can follow control flows within the already known regions and discover any missing code, as well as data in between and in the middle of functions. After analyzing 491 ELF binaries, we have found a few cases of data embedded in code. However, in most of these cases, the gap region indicated in the `.eh_frame` section was simply the padding data in between function or section boundaries. There were only a few

name	total code ptr	return ptr	jump table ptr	exception handler	exported function	percentage remapped
400.perlbench	17548	14101	1542	0	3	89%
401.bzip2	512	365	48	0	2	80%
403.gcc	58264	47847	6496	0	7	93%
429.mcf	146	94	0	0	2	66%
445.gobmk	12220	9245	266	0	3	78%
456.hmmer	4426	3624	223	0	2	87%
458.sjeng	1436	1124	140	0	3	88%
462.libquantum	585	448	1	0	2	77%
464.h264ref	3738	3059	89	0	3	84%
471.omnetpp	21116	16700	956	3708	23	94%
473.astar	535	411	3	0	2	78%
433.milc	1881	1561	38	0	2	85%
435.gromacs	8491	6936	321	0	19	86%
437.leslie3d	693	631	0	0	2	91%
444.namd	1343	1157	8	16	3	87%
447.dealII	48927	37679	2801	6632	10	89%
450.soplex	6668	5237	570	493	5	91%
453.povray	13887	10559	1702	103	26	89%
454.calculix	19318	17699	206	0	2	92%
470.lbm	126	79	0	0	2	64%
482.sphinx3	2930	2530	5	0	2	87%
libc.so.6	26719	12117	12432	0	2163	98%

FIGURE 4.3: Percentage of Randomized Pointers in SPEC Benchmark

Name	`.eh_frame` Coverage
spec2006	97.54%
libc.so.6	97.87%
libm.so.6	96.16%
libgfortran.so.3	98.58%
libquadmath.so.0	99.63%
libstdc++.so.6	95.44%
libcrypto.so.1.0.0	87.23%
average	97.17%

FIGURE 4.4: Coverage of exception handling information on the original code.

.

cases in which data was embedded in the code as part of jump tables. Figure 4.5 provides details about these cases.

We has found 40 locations in total 390 bytes where there is data in `libc.so.6`, all of them used as padding. Since the value of the padding is zero, they can

Name	Invalid Region	Valid Region	Reason
libc.so.6	40	0	alignment padding
libffi.so.6	0	1	ffi_call_SYSV
libcrypto.so.1.0.0	0	16	lookup table for crypto algorithms

FIGURE 4.5: Embedded data regions identified by static analysis results

cause disassembly errors if not handled properly. In `libffi.so.6` on x86, we found a jump table in the middle of code inside the `ffi_call_SYSV` function. The same library on x86-64 has two jump tables identified by this algorithm. Finally, `libcrypto.so.1.0.0` contains 16 data regions in the middle of code in both 32-bit and 64-bit versions. In particular, we further looked into the binary and retained 20377 Bytes of data in the code segment. All the data regions are located after function return.

4.7.3 Strength of randomization

To evaluate our randomization protection on instrumented code, we port our code into a dynamic loader[8] running in 64 bit Ubuntu 14.04. In this experiment, we tested chrome 43.02 by forcing our modified loader to load the browser code as well as its dependent libraries. In the experiment, we used instrumented code of the same size as the code in an original binary,[9] i.e., when a module is loaded, our loader immediately load a instrumented code piece whose size is the same as the text segment of the module. In our experiment, all 24 processes of chrome has been tested.

Our experiments illustrate that the size of instrumented code used by chrome is 514 Megabytes. The instrumented code pages allocated are scattered in the whole user address space. The address range is different on each process, but the length of address space that instrumented code occupied across different processes is between 953 and 1021 Tera-bytes. Since instrumented code is not targeted by any code pointers, the probability of memory leak is calculated by size of code divided by the address space used, which is between 5.3e-7 and 5.68e-7.

[8]This dynamic loader belongs to glibc 2.19

[9]This might introduce a tiny inaccuracy since instrumented code size is slightly larger than original code. However, given the fact that the address space is several order of magnitude larger, this variance could be ignored.

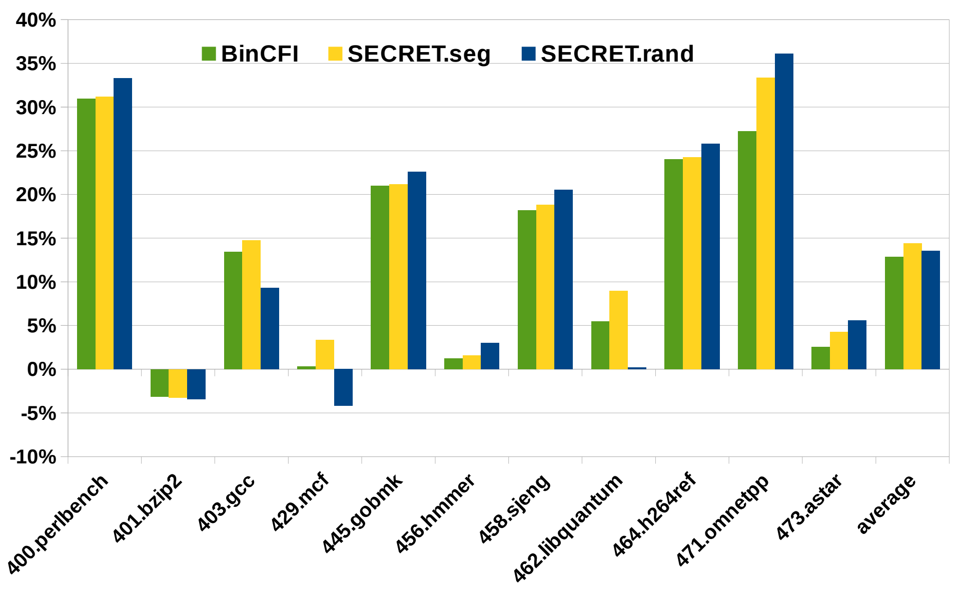

FIGURE 4.6: Runtime overhead of SECRET on the SPEC benchmarks.

4.7.4 Runtime performance

We have evaluated SECRET's runtime overhead using the SPEC2006 benchmark. Since code space isolation does not introduce extra overhead, we include this feature on default except in baseline system PSI. In our experiments, three modes of our prototype has been tested in three modes: 1) `PSI`: baseline protection; 2) `SECRET.seg`: instrumented code protected by memory segmentation; 3) `SECRET.rand`: instrumented code protected by randomized base address. In all cases, SECRET transforms the main executable and all 6 dependent libraries. The results of each of the SPECINT benchmarks are shown in Figure 4.6, while Figure 4.7 shows the average overhead for SPECINT and the total for all 21 SPEC CPU benchmarks.

In the `SECRET.seg` mode, the average runtime overhead for SPECINT is 14.41% (the total SPEC CPU overhead is 15.64%). In this mode, both the instrumented code and its LTT are located outside of the memory sandbox. The overhead in this mode mostly comes from memory access through a segment register when performing address translation. In the `SECRET.rand` mode, the average runtime overhead for SPECINT is 13.54% (the total SPEC CPU overhead is 14.48%). The protection added in this mode is the base address randomization of instrumented code. Compared with `SECRET.rand`, there two differences in this mode: (a) the address translation trampolines are required to do two range checks (one for original code space and the other for randomized code address space) instead of one, this

	PSI	SECRET.seg	SECRET.rand
SPECINT	12.84%	14.41%	13.54%
TOTAL	14.20%	15.64%	14.48%

FIGURE 4.7: Summary of SPEC2006 Runtime Overhead

Test suite	Base	SECRET.rand	Description
python	4.709	5.022	run bincfi script to transform /bin/ls
dd	99.46	99.6	copy a 1GB file
md5sum	2.44	2.45	checksum of a 1GB file
apt-get	1.54	3.6	update source list
scp	2.78	2.96	copy a 100MB file to server

FIGURE 4.8: Completion time (sec) for real-world programs.

will add a little overhead and (b) `SECRET.rand` does not require intensive memory access through our TLS. Our experiments illustrate that the average overhead of `SECRET.rand` is slightly lower than `SECRET.seg`. Compared with the baseline system PSI, the average runtime overhead added by `SECRET.rand` is less than 2% and `SECRET.rand` less than 1%.

In addition to the SPEC benchmarks, we also evaluate SECRET with several real world programs. As the SECRET.rand mode includes all features and represents programs in current architecture (x86-64), we use this mode to compare with the performance of the original programs. The results of Figure 4.8 show that SECRET is practical for real world usage. This experiment includes script interpreters such as `python` and `perl`, disk I/O tools such as `dd`, as well as network related tools such as $\texttt{apt} - \texttt{get}$ and `scp`. In all experiments, the code of all main executables and libraries was transformed to instrumented code.

4.7.5 GUI program startup overhead

One of the important overheads covered in the SPEC benchmark is the startup overhead. SECRET has noticeable startup overhead because of our modified loader needs to do the following things for each code module: (a) loading instrumented code as well as LTT; (b) initializing the entries in GTT, for code pointer remapping to work properly and (c) wiping out original code immediately after load time.

Name	Base (sec)	PSI	SECRET.rand
evince	0.34	135%	168%
gcalctool	0.62	110%	161%
gedit	0.6	120%	165%
vim	0.6	60%	67%
lynx	0.02	100%	100%
LibreOffice	1.4	51%	200%

FIGURE 4.9: SECRET's startup overhead on GUI programs.

To accurate evaluate this overhead, we intentionally use GUI programs since they will load much more library code than benchmark programs. This helps improving the accuracy of startup overhead evaluation. As in the previous experiment, we use the SECRET.rand mode to compare with the original program and the baseline (PSI).

Figure 4.9 illustrates the startup overhead of several well-known Linux applications, including three GTK and two text user interface programs. The results show that SECRET's overhead is slightly higher on GTK programs than text user interface programs, because more libraries are loaded at program start up.

4.8 Discussion

4.8.1 Harvesting return addresses

In many scenarios such as JIT environments, attackers may have crash-restart ability to perform repeated testing. In these cases, attackers may intentionally launch native functions to harvesting return addresses (return addresses will be generated by the caller of the native functions and themselves due to invocation of subroutines. Note that code space isolation randomizes useful code pointers from attackers that try to recover other code pointers. These encrypted code pointers are further protected by code space isolation. However, function pointers remain unchanged, i.e., they stay in the form of original code. When any of these functions are invoked, the memory footprint that this function left remains on top of stack. The footprint may include some of the return addresses when subroutines of this function are invoked.

Despite that those return addresses are still encrypted, knowledgeable attackers may take advantage of them. Since attacker is knowledgeable on the original binary, she knows which function she invokes and she could fully understand the function internal structure by looking at original code. Then she could gather all the callsite gadgets by invoking those functions on which she has prior knowledge. Even if the return addresses are randomized, attackers could easily use them.

We argue that to prevent this memory disclosure, SECRET could take a simple approach: *eliminating all used return addresses*. In particular, we could eliminate the return address left on the stack immediately after we start translating the return address. This method will not cause compatibility issue because when a return address is used by an return instruction, its location is already on top of stack. Note that this location won't be used by programs, since any value on top of stack may be overridden due to signal delivery at any time. By doing so, no function gadgets can be used to infer additional callsite gadgets.

4.8.2 Call-oriented programming

As described in Section 4.4, our system does not change function pointers. This is reasonable because in static binary instrumentation, changing constants that look like function pointers is unsafe because those values could be used as integers. Unfortunately, this allows attackers to use all original function pointers to launch a call oriented programming (COP) attack such as recent work COOP [120] does. We admit that this is the limitation of our current prototype. However, we argue that this limitation could be minimized once we start doing static analysis focused on C++ programs such as vtable analysis. Besides, in order to to successfully launch a COOP attack, the attacker has to fulfill lots of requirements. For instance, COOP requires a certain code piece that is a for loop containing a virtual function call. In addition, the pointer to traverse the object list is required to be corrupted. Blindingly looking for such a code piece without reading code pages is expected to be challenging.

4.9 Summary

Defending against advanced code reuse attacks that take advantage of memory disclosure vulnerabilities is becoming increasingly important. To that end, breaking the ability of attackers to read the executable memory segments of a process, or even to infer the location of potential gadgets, can be a significant roadblock for attackers.

In this chapter, we have achieved the above goal by implementing our system SECRET. SECRET combines two novel code transformation techniques, *code space isolation* and *code pointer remapping.* The former prevents read accesses to the executable memory of the instrumented code, a protected version of an application's original code, while the latter decouples its required code pointers from that of the original code. We have demonstrated that SECRET, the prototype implementation of the proposed concept, can offer practical and comprehensive protection for real-world COTS applications, by combining disclosure-resistant code and coarse-grained CFI with reasonable performance overhead.

Chapter 5

Application: Comprehensive Protection From Code Injection Attacks

In previous chapter, we have discussed our practical defense against code reuse attacks. However, in principle, it is unclear whether code reuse attacks could be fully defeated on COTS binaries due to more advanced attack styles such as call oriented programming (COP) in COTS binaries.

To remedy that, in this chapter, we extend PSI with an additional security property: **code integrity** to defend against native code injection, attacks that introduce malicious code into vulnerable applications. We argue that native code injection is the ultimate goal of all these advanced code use attacks. Preventing native code injection will effectively mitigate these attacks.

We will demonstrate that code integrity when combined with control flow integrity, is an effective approach against modern code injection attacks. The following discussion will illustrate how this security property is enforced and implemented.

5.1 Motivation

Despite decades of sustained effort, memory corruption attacks continue to be one of the most serious security threats faced today. Memory corruptions are sought after by attackers as they provide ultimate control — the ability to execute low-level code of attacker's choice. This factor makes them popular in targeted as well as indiscriminate attack campaigns.

The popularity of data execution prevention (DEP) stems from its ability to block the highly sought-after arbitrary code execution capability. This is one of the reasons why it became popular despite its well-known weakness against code reuse attacks such as return-to-libc [101] and Return-Oriented Programming (ROP) [126]. Subsequent to its deployment, attackers have become increasingly skilled at crafting ROP attacks that string together small snippets of existing code (called "gadgets") into something like a virtual instruction set. Although researchers have achieved Turing-completeness with ROP attacks, real-world ROP exploits face many serious limitations, including:

- *Limited variety of gadgets.* Due to defenses such as ALSR, frequent software updates, customized compiler optimizations, version changes and others, some gadgets are eliminated, while the location of others is unknown, and requires significant effort to locate. This limits the range of attacks possible.
- *Payload sizes are typically limited.* The payload size that attackers can use to put their ROP chain is usually decided by specific exploit context. Recent research [69] shows the payload size limit imposed by vulnerabilities in `Metasploit Framework` [117], one of the best security testing tool that provides various exploit samples and payloads. This framework contains The average of maximum payload among 946 vulnerabilities covered in the framework is only 1332 bytes. This poses constraints on the length of ROP chain.

For this reason, real-world ROP attacks have been relatively short, and have targeted executing just enough code to disable DEP, i.e., change permissions on memory pages, or loading their own code pages or libraries. Such a transition from code reuse to code injection (CRCI) has become a common feature of today's exploits.

Base on the above analysis, it is believable that in the foreseeable future, attackers will continue to be constrained in their ability to craft pure ROP exploits. Even as attackers make advances in ROP attacks in the future, platform vendors are bolstering defenses, e.g., by increasing ASLR entropy using 64-bit address spaces, while processor and/or OS vendors consider including ROP-oriented defenses [78, 96]. On the other hand, launching a small ROP that disables DEP is considerably simpler than a pure ROP exploit. Moreover, once DEP is disabled, injected payload can start execution, and it is no longer constrained by ASLR. In contrast, for a pure ROP attack, its susceptibility to ASLR increases with each additional gadget it needs to use. For this reason, native code injection attacks will continue to be the target of attackers for the foreseeable future. Consequently, systematic defenses against code injection attacks can greatly degrade the capabilities of attackers. This is the goal of this chapter. Specifically, we propose a strong code integrity approach that ensures that:

- a process can only execute native code that it is explicitly authorized to execute, and
- even an adversary that hijacks all executing threads of a victim process cannot defeat this property.

This strong defense against injected code attacks is achieved without sacrificing compatibility with existing software, the need to replace system programs such as the dynamic loader, and without significant performance penalty.

5.1.1 Property and Approach

Existing code integrity mechanism realized by DEP and write protection on code is targeted at native code injection attacks, but is secure only against a weak attacker model. Attackers may introduce new code by injecting their own malicious modules or changing page permissions to mark their payload executable, or even hijacking the loader. These could be easily achieved by a code reuse attack, which is quite mature as shown in recent research [62, 74] as well as real world exploits and vulnerabilities [18–22]

The fundamental reason is however, not the power of code reuse attack, but instead the missing security mechanism that ensures code integrity. Code signing is one of the techniques that protects code at launch time. Unfortunately, once code is loaded, it may still be modified after passing the code signing check.

To solve the issue, we therefore propose a practical but strong code integrity. Specifically, this approach enforces the following:

- Any given program (binary) can load only a specified set of files containing executable code.
- Only instructions legitimately contained in these files can ever be executed.
- No unauthorized changes can be made to any of the instructions during runtime.

Together, these properties make native code injection impossible. Note that techniques such as write-protecting binary files (or code-signing) serve to preserve the integrity of code on the disk, but don't address the protection of code integrity in main memory. This approach thus provides an important missing piece to ensure end-to-end code integrity.

This approach is called as Strong Code Integrity (SCInt), highlighting the fact that it ensures the integrity of all code, together with control-flows within the code. This contrasts with existing CFI techniques that do not consider most attacks on code integrity:

- CRCI attacks can modify memory protection to either overwrite existing code or make non-code [1] executable.
- Loader subversion attacks can load malicious code by exploiting vulnerabilities in the dynamic loader.

A more complete list of possible attacks thwarted by SCInt appears in Section 5.2.

While SCInt aims to ensure code integrity, control flow integrity as a underlying feature ensures that none of the policies enforced by SCInt can be bypassed. Code integrity property, in turn, protects control-flow integrity by preventing CFI checks from being bypassed by newly introduced native code.

5.1.2 Key features

The implementation achieves code as well as control-flow integrity without restricting applications, or requiring the replacement of system software such as the system loader or the standard libraries. Specifically, the contributions are:

[1] i.e., data pages containing attacker-provided payload

- *Describe many possible ways to inject native code despite modern protections such as DEP and ASLR.* Attackers may introduce new code by injecting her own library or changing existing code by hijacking the dynamic loader. Many of these attacks have not been discussed in previous work, while others have been known and reported in CVE reports. However, defense to them has not been discussed.

- *A secure library loading/unloading state model* that ensures that every code segment is correctly identified and mapped for execution, and that these code segments are never writable. An important aspect of this design is the simplicity of this model, which increases confidence that it correctly enforces policies to ensure secure loading.

- *Defense against native code injection attacks by a powerful adversary.* This defense is secure against attackers that have defeated ASLR, and are able to compromise one, two, or all execution threads using code-reuse attacks.

- *No need to trust the loader.* Defense against code injection attacks is achieved *without* assuming that the loader itself is secure, or that it can't be exploited. As with any large and complex piece of software, it is unrealistic to expect all vulnerabilities to be eliminated from the dynamic loader. The novelty of this design is its ability to protect against loader exploits and subversion attacks using a simple state model.

- *Efficient policy implementation.* We present techniques to speed up policy enforcement. Overall, this approach adds a small overhead over the base platform (BinCFI) for application start-up, and very low overheads at run-time.

5.2 Possible code injection vectors

The threat model considers remote attackers that are able to interact with network-facing running application programs. We assume that such attackers can exploits vulnerabilities in the applications to:

- using information leakage (or other) attacks to defeat ASLR,

- hijack the control flow of *more than one* thread in the victim process, and

- read/write/execute arbitrary memory of the victim process, subject to page protection settings by using code reuse attacks to load new code, e.g., malicious or vulnerable libraries.

Below, we describe concrete or possible instances of attacks that may be carried out by such attackers.

5.2.1 Direct attacks

In this case, the exploit code (typically an ROP payload) directly invokes the necessary system calls such as `mmap` to map new code into memory, `mprotect` to change execute permissions and `exec` to launch executables. Since most operating systems do not block these system calls, (legitimate programs use them to load libraries), many current exploits rely on this approach.

5.2.2 Loader subversion attacks

Successful code injection attacks can be launched even if countermeasures are deployed against direct attacks: even if privileges relating to code loading are taken away from the application code, the loader code that is part of the process needs to be able to exercise these privileges. Attackers can thus gain these privileges by subverting the loader.

5.2.2.1 Control hijacking attacks

These are the most direct form of loader subversion: simply invoking functions within the loader for mapping memory pages for execution, or making executable memory writable.

Loading malicious libraries If an attacker has previously stored a malicious library on the victim system, then she can use a code reuse attack to load this library.

There is a common misconception that statically linked binaries are immune to such attacks. In reality, even statically linked code on Linux needs some dynamic loading capabilities to perform start-up initialization such as TLS (thread-local storage) setup, stack cookies, and parsing vDSO[2]. Hence they contain some internal functions such as `_dl_map_object`, `_dl_open` and `_dl_open _worker` that can be utilized by an attacker to load malicious libraries.

Turning on stack executability A common assumption for many security defenses is the existence of DEP, which is also the first obstacle for attackers. However, on Linux, this assumption can be invalidated by invoking a function `_dl_make_stack_executable` in the loader[3].

5.2.2.2 Data corruption attacks

Library hijacking attacks Executable files specify the libraries they depend on, but not the search path used to locate these libraries. Recent vulnerability reports [7–9] indicate that by subverting the search path, attackers can load malicious libraries. Search path can also be controlled using environment variables such as `LD_PRELOAD`, `LD_AUDIT`, `LD_LIBRARY_PATH`, or search path features like: `$ORIGIN`, and `RPATH`. Equally important, memory corruption attacks can modify search path related data structures, thus overriding the original path setting.

Leveraging these attack vectors, attackers can load unexpected libraries in place of original libraries [2, 12]. This may lead to privilege escalation attacks [5, 6, 10, 11]. On Windows, FireEye reports [134] an increasing use of the WinSxS side-by-side assembly feature [25] to load malicious libraries, bypassing normal search for libraries in typical directories containing DLLs.

Malformed ELF binaries Like any complex piece of software, the dynamic loader is bound to contain vulnerabilities [3]. Malformed binaries are one of the best ways to trigger them [4, 13]. It has been reported that malformed relocation data can be used to circumvent code-signing on certain Apple iOS versions [15]. Researchers have also documented more general attacks, showing how malformed relocation information can be utilized to perform arbitrary operations [128].

[2]vDSO is a dynamic library exported by the Linux kernel to support fast system calls.

[3]Stack executability is a "feature" that is available to support GCC nested functions, a legacy feature.

Corrupting loader data Some binaries require the loader to perform relocation. It is possible to exploit this capability to modify existing code. In particular, the author found an attack that first corrupts the relocation flag, then replaces the relocation table and symbol table with forged versions by overwriting loader data structures in memory. A subsequent code reuse attack, involving a call to the loader function for performing relocation, resulted in a successful modification of existing code.

Note that this attack is available not only on glibc, but also other loader implementations such as those packaged with bionic libc for Android (before 4.3), uClibc for embedded systems and other POSIX-compliant libc such as musl-libc.

5.2.2.3 Attacks based on data races

Several potential opportunities exist through which an attacker-controlled thread can modify loader data while it is being used and/or modified by the loader. We have identified three attractive avenues in this regard:

- *File descriptor race:* In order to load a library, the loader first performs an `open` on the file to obtain a file descriptor fd, and then uses fd in an `mmap` operation to map the code and data pages into program memory. An attacker's thread can race with the loader to change the file pointed by fd. This can be accomplished using system calls such as `dup2`.
- *Racing to corrupt data segments used during loading:* Data segments in the library are loaded with write-permission enabled. An attacker's thread can race with the loader to corrupt parts of this data that contain ELF segment information. When this corrupted data is used by the loader to load code segments, the loader may end up doing the attacker's bidding.
- *Racing during relocation:* Binaries that rely on text relocation provide another opportunity. Specifically, during the time of relocation, the loader maps the executable pages for writing. An attacker's thread can now overwrite the code being patched by the loader.

5.2.3 Case study: code injection using text relocation

This section describes an interesting attack that allows code injection by leveraging the loader code and data using text relocation.[4]

Note that code sections are normally write protected in ELF executables, thus preventing attacks that overwrite them. Text relocation is a convenient key to open the "door." Although text relocation is discouraged since it makes it difficult to share code memory across processes, it is still used in some libraries and its code is available in all versions of glibc loader.

The attack is launched simply in the following steps:

- Bypass ASLR and figure out loader data structure
- Corrupt loader data structure
- Transfer control to loader function

5.2.3.1 Bypassing ASLR

The first step is to by pass ASLR and figure out loader data structure and text relocation function inside loader. Dynamic loader contains a data structure called `link_map` for each code module. This data structure stores the metadata information such as the address of symbol table, address of relocation table and whether the module has been processed, as shown in Figure 5.1.

There are several methods to figure out the memory location of `link_map`. The first one is to check the Global Offset Table (GOT). In fact the 2nd element of GOT (GOT[1]) contains the address of `link_map` in most of binaries compiled with partial RELRO. In case for binaries that are compiled with full RELRO or binaries launched with eager binding (with parameter `LD_BIND_NOW`, GOT[1] is not initialized.

However, none of these counter measures stop an attacker. The experiment shows that in program binary, there is an special memory segment called `.dynamic`, which contains information of the binary at runtime. In particular, there is an entry

[4]text relocation is used by dynamic loader to patch code pointers in code segment at runtime to cope with ASLR.

called DT_DEBUG. This entry points to a data structure that contains a pointer to the address of link_map.

Although it is easy for dynamic loader to disable DT_DEBUG by simply eliminating 2 lines of its source code, such code modification will be unlikely, since it poses inconvenience for program debugging.

Once link_map address is known, it is easy to figure out text relocation function address. This is because link_map is stored as an array, while the 1st one is for the executable and one after is for the dynamic loader. From the 2nd link_map data structure, the base address of loader is known. Then attackers can simply use binary scanning to find out the function address.

5.2.3.2 Corrupting loader data structure

When the memory address of link_map is leaked, this work leverages a memory corruption attack to corrupt data pointers of relocation table and symbol table to point to attacker's payload. In addition, this work modifies a flag in link_map for text relocation patching. Specifically, this flags indicates loader to unprotect write protected code pages. and this flag is corrupted into value DT_TEXTREL as shown in Figure 5.1. The the memory address specified by the crafted relocation will be updated and by the value specified by the crafted symbol table. This way attack could write to arbitrary code location with any value.

The link_map data structure shown in figure 5.1 contains related metadata for this attack, such as pointers for relocation table (l_info[DT_REL]), string table (l_info[DT_STR]), symbol table (l_info[DT_SYMBL]). These pointers do not directly point to their data but all point to the locations inside the read-only dynamic section (when RELRO is applied). This does not prevent attacker, since they could corrupt the data pointer to attackers' payload. In addition, to launch a successful attack, l_relocated is a flag that should be flipped, since it indicates whether the object has been relocated. Finally, GOT needs to be taken over too, because, the function used will initialize GOT table. However, after relocation patching, the .got section in ELF file become read-only, this will trigger a segmentation fault. Changing the GOT to any arbitrary location that contains twelve bytes of writable memory should work (3 GOT entries).

```
struct link_map
{
  ElfW(Addr) l_addr;  /* Base address  */
  ElfW(Dyn) *l_ld;    /* Dynamic section */

  ElfW(Dyn) *l_info[DT_NUM];

  /* l_info[DT_TEXTREL] = 1
   * l_info[DT_REL] = attacker_rel;
   * l_info[DT_SYMBL] = attacker_sym;
   * l_info[DT_GOTPLT] = attacker_got;
   */

  const ElfW(Phdr) *l_phdr;  /* program header*/
  unsigned int l_relocated:1;
};
```

FIGURE 5.1: Dynamic Loader Internal Data Structure

5.2.3.3 Invoke text relocation function

When the link_map data structure is corrupted, code injection attack can be launched by invoking an internal function (`_dl_relocate_object`).

(`_dl_relocate_object`) takes 4 parameters, the first parameter is the address of `link_map`. The 2nd one is not used. The third one is the relocation mode, here, the value can be simply 0x1 (representing `RTLD_LAZY`). And the last one indicates whether doing profiling or not. Again, pass an integer 0 is fine.

To simplify the prototype, the author uses a simple program that contains a buffer overflow which allows attacker to change the content of stack including return addresses. Using this exploit, the author launches the text relocation function in one shot. Further, the author modify the stack frame pointer to make sure when function returns, it goes to the injected payload.

This runtime relocation attack used in above case can only corrupt the current module of the link_map data structure. However, it can be generalized to corrupting all memory region in the runtime. This is true if attackers work harder to corrupt one more pointer, l_phdr. l_phdr points to the ELF program header table located in the write protected ELF image. Corrupting this data pointer allows attacker to fool dynamic loader to unprotect and corrupt arbitrary code or data.

5.3 System design

To defeat the kind of attacks described in the previous section, we propose a security primitive for code loading which ensures that a process executes only the native code that it is explicitly authorized to execute. The design ensures the integrity of all code from the binary file to its execution.

Note that the design of SCInt is independent of underlying CFI. However, for the purpose of explanation and demonstration, the implementation is based on PSI [1, 153].

SCInt secures all operations related to loading and manipulation of code. It is able to do this *without requiring changes to the loader.* Instead, it relies on a small reference monitor that intercepts key operations relating to code loading, and ensures their safety. This reference monitor is based on a state model described below.

5.3.1 State model

The state model captures the essential steps involved in loading a binary and setting up various code sections contained in it. It does not rely on non-essential characteristics that may differ across loaders, and hence is compatible with different dynamic loaders, including eglibc, uClibc, musl-libc and bionic-libc. The key operations performed by most dynamic loaders on UNIX are:

- *Step 1:* Open a library file for read.
- *Step 2:* Read ELF metadata. (This metadata governs the rest of the loading process.)
- *Step 3:* Memory-map the whole ELF file as a read-only memory region.
- *Step 4:* Remap each segment of the ELF file with the correct offset and permission.
- *Step 5:* Close the library file.

Calls to system functions used by the loader to perform these operations are rewritten by SCInt so that they are forwarded to the state model, which checks these operations against a policy, and if permitted by the policy, forwards them to the original system functions. All checks are performed using binary instrumentation. Note that the underlying CFI enforcement ensures that none of these checks can be bypassed.

Note also that the policies need to maintain some state, and this state needs to be protected from attacks by compromised execution threads within the vulnerable process. We describe in Section 5.3.6 the design of this protected memory.

Figure 5.2 illustrates the state model and summarizes the enforcement actions in each step of the model. This state model ensures the following properties:

- Only allowed libraries can be loaded into memory address space. These libraries may be specified using their full path names. Alternatively, the policy could permit loads from specified directories.
- Each segment in the module must be loaded in the correct location as specified in the ELF metadata. (Thus, the policy enforcement must be capable of reading and interpreting the ELF header.)
- An executable segment is never mapped with write permissions. Moreover, any memory page that was ever writable will never be made executable.
- No two segments can overlap, nor can there be an overlap between a segment and any previously mapped (and still active) memory page.

5.3.2 State model enforcement

SCInt maintains the current state of an ongoing load. A loading related operation is allowed only if the state model is in the state where that operation is legal. For simplicity, the state model serializes file loading, i.e., one library cannot be loaded until the completion of loading of a previous library. The state model handles some common errors that can occur during a file load, such as errors in opening of files, obtaining enough memory and/or address space for mapping, etc.

5.3.2.1 Operation to open a library

SCInt intercepts calls made by the loader to `open` files for the purpose of loading libraries. It first copies the file name into protected memory, and this copy is

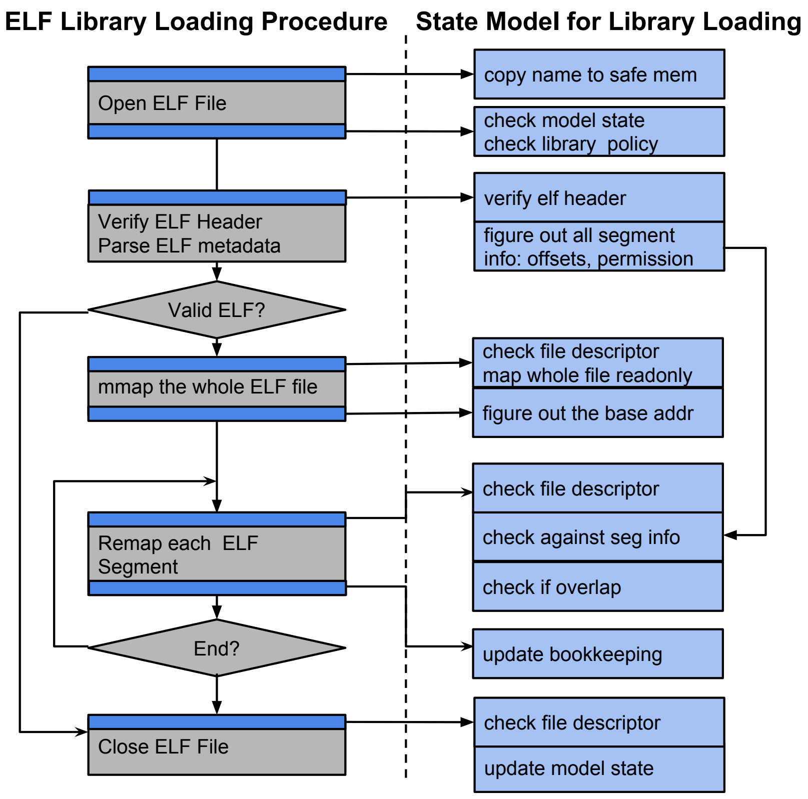

FIGURE 5.2: State Model for Module Loading

passed onto the system call to preclude TOCTTOU attacks. The actual check on file name validity is deferred until the file open operation returns with success. At this point, the file descriptor value is also copied into protected memory for use in subsequent stages of the state model.

Ideally, library policy checking should ensure that all libraries loaded are from a predefined set. However, in practice, it is not always easy to determine the exact set of libraries needed by an application in advance, as libraries may rely on runtime information in deciding which libraries to load. This is particularly true for many graphical programs and plugins. To simplify policies for such applications, SCInt can be configured to permit loading of any library from a set of specified directories such as `/lib` and `/usr/lib/*`. It is also possible to tighten the policy for specific libraries, such as `libc.so`, so that they are loaded from a specific file, or a specific directory.

The underlying platform enforces a policy that all loaded libraries be transformed for CFI. It provides an on-demand transformation of binaries when a loading of untransformed binary is attempted.

5.3.2.2 Operations to map file into memory

Note that `mmap` operations that load libraries into memory are based on file descriptors rather than file names. A table in protected memory is used to maintain these associations, and is populated by the state model at the end of Step 1. Any attack that invalidates this association can compromise the library loading policy, and hence SCInt guards against such invalidation. Note that a valid file-descriptor-to-file association can be changed as a result of the operations `close`, `dup2` or `dup3`. In fact, the `dup` operations have the effect of closing a file descriptor too, so this work effectively needs to handle just the case of closing of a file descriptor. The state model deletes filename/file-descriptor pairs on such close operations. Any future uses of that descriptor by the loader in `mmap` operations will be denied by the state model.

5.3.2.3 Segment boundaries

Segment boundary checking ensures that library code is loaded into memory as intended. Segment information is parsed in Step 2 of the state model. The read operation made by the loader to input this information is intercepted and modified so that its results will be stored into protected memory. SCInt then parses this ELF metadata to obtain information about segments and where they should be loaded. At this point, the header is copied back into the read buffer provided by the loader so that the loader may continue with subsequent steps in loading.

In Step 4, the information saved about segment offsets will be used to validate requests to map segments of the library into memory. In particular, SCInt ensures that each code segment is mapped at the offset specified in the ELF header, it is never mapped with write permissions, and that the segment does not overlap any other segment. As a result, even if attackers corrupt the loader's in-memory data structures holding ELF metadata, they will not be able to circumvent SCInt. In particular, they may fool the loader to request mapping with incorrect permissions (e.g., code segment with write permissions), but this request will be denied by the state model because it is inconsistent with the metadata read earlier by SCInt and stored in protected memory. Thus, such attacks will be detected and stopped by SCInt.

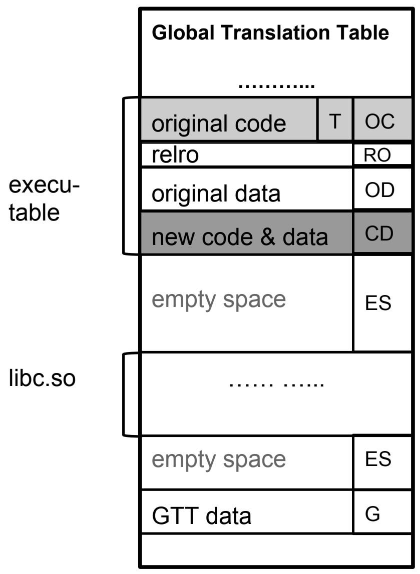

FIGURE 5.3: Layout of Memory Map Table

5.3.3 Code integrity enforcement

While the state model ensures that code is safely loaded from disk to memory, additional steps need to be taken to ensure continued code integrity throughout the execution of a process. SCInt achieves this by maintaining information about memory segments, and enforcing policies on operations that attempt to change these mappings or their associated permissions.

In particular, SCInt put the list of memory segments that belong to code modules into a memory map table. Each memory segment in this memory map table will be labeled with the following tags:

- *Tags to indicates different segments* such as the original code,[5] data, and transformed code. One tag is added for each segment. For convenience, address space that does not belong to any module will have a tag with value MEM_EMPTY.
- *Feature flag or state flag* which is included in some of the segments to indicates whether there is code relocation, etc.

[5]Recall that BinCFI and many other binary transformation/translation systems such as Pin [90], DynamoRIO [51] and Reins [141] keep the original version of the code as is, while the instrumentation is performed on a copy. Specifically, CFI instrumentation is inserted by BinCFI into the transformed code, while the original code is preserved as is, and made readable but not executable.

Figure 5.3 demonstrates the case of a SCInt transformed executable file image. It consists of several segments, including the original code, and the original data. Transformed code and data is appended at the end of the file. Note that only transformed code is executable. SCInt enforces a consistent segment permission with the specification in the ELF metadata.

It is worth mentioning that "`relro`" is a special data region that is first made writable by the loader. It is then "patched" by the loader and then its permissions changed to read-only. This section usually contains important code pointers and data pointers that the loader wants to provide an extra level of protection by making it read-only. SCInt tracks this information and protects the segment as described below. To prevent attackers from corrupting running code, data or introducing executable code, SCInt uses a policy to constrain system operations by checking against the memory map table in Figure 5.3. The following shows the policy on memory operations.

- **Original Code (OC)**: No operations are allowed.
- **Original Code with Text relocation (OCT)**: Original code with text relocation can be unprotected once, followed by one-time mprotect with read and execute permission. No further operations are allowed afterwards. Note that this means that there is a window of time during which original code may be modified by an attack, but this does not affect the security of the scheme because the underlying CFI (here, BinCFI) ensures that control flow never goes to the original coded.
- **Relro Data (RO)**: relro data can be memory protected so that it is read-only. No other operations are allowed.
- **Original Data (OD)**: No operations are allowed to make the target region executable, unless a special variable is setup (details in Section 5.3.7). Everything else is allowed.
- **New Code and Data (CD)**: No operations are allowed on transformed code or transformed data.
- **Virtual DSO (VD)**: Virtual DSO is used to support fast system calls. No operations are allowed on this segment.
- **Empty Space (ES)**: No operations are allowed to make target region executable.

	OC	OCT	OD	CD	RO	VD	G	ES
None								Y
R	Allow	Allow			Allow			Allow
RW		Allow [a]			Allow [a]			Allow
W		Allow [a]			Allow [a]			Allow
WX		Allow [a]	Deny [b]					

[a] permission can be used for only one operation on this segment.
[b] request denied if not configured for unsafe JIT code

Figure 5.4: Permitted protection settings for memory segments

- **Memory Map Data (G)**: No operations are allowed on memory map data.

The above list shows the policy on different memory regions. Figure 5.4 illustrates the policy for `mprotect` system call in a tabular form.

5.3.4 Library loading policy

Library loading policy ensures that each executable can only load a set of dependent libraries. This dependent library set is decided ahead of time. Enforcing this library loading policy may cause two kinds of usability issues:

- Identifying the set of all required libraries can be difficult, both due to the large number of libraries loaded by many applications, and because this list can vary across localities and configurations.
- Tasks such as debugging require a different set of libraries to be loaded.

To figure out potential dependencies, this work expects the constraint to be that applications can load libraries only from certain standard locations. Heuristics to get those locations could either come from the initial set of explicit dependencies (obtained using a binary tool call `ldd`), or by profiling. Further tightening to specific libraries may be useful for some high-value targets such as browsers and pdf readers, where a higher level of effort to list libraries would be justified.

To support tasks such as debugging, it is often necessary to use environmental variables such as `LD_LIBRARY_PATH` and `LD_PRELOAD` to change the loading path or add additional dependencies. The policy is flexible enough to handle these needs, since different policies can be applied to the same application run by different users and/or in different running environments. It is practically achieved by intercepting code loading operation in `ld.so`. `LD_LIBRARY_PATH` and `LD_PRELOAD` are

```
Original Code :
  mov $const, %eax
/ * REL : 0x8048346 : R_386_32 * /
  ......

Transformed Code :
  _8048345 :
  call_next
  _next :
  pop%eax
  add$offset, %eax
  movl(%eax), %eax
```

FIGURE 5.5: Transformation for Text Relocation based Code Patching

fully supported in SCInt. However, only valid values of LD_LIBRARY_PATH should be allowed by administrators who design the policy, since LD_LIBRARY_PATH itself could cause security issues (CVE-2012-0883).

5.3.5 Compatibility with code patching

There are legitimate reasons to modify code after it is loaded, e.g., text relocation. Text relocation allows programs to run in any memory location by updating the code pointers at runtime. This code patching does not violate the policy since only (non-executable) original code will be patched. However, due to incorrect patching location, transformed binaries won't be correctly executed. To cope with it, this work use a special code transformation for all instructions that use text relocation, as shown in Figure 5.5. In particular, this work uses position independent code pattern to fetch the address of the constant in original code. When text relocation is patched at runtime, transformed code uses this code pattern to get the "patched" value. By doing so, transformed code with text relocation could be correctly executed.

5.3.6 Protected memory

Our protected memory on x86-32 is realized using the hardware segmentation mechanism supported on this architecture. Several static instrumentation approaches have been developed for accomplishing this on the x86 architecture [95, 146, 148], but all of them have required some level of compiler support. In contrast, we present a compiler-independent solution that works on COTS binaries. Another key benefit of the solution is its support for thread-local storage (TLS). TLS support is either lacking in previous implementations [70], or requires compiler support [146].

At the time of loading an executable, SCInt reserves a region of memory that is to be protected by memory segmentation. Specifically, we set aside the top 128 MB of the lower 3GB of the address space for memory segmentation protection.[6]

Reserving the highest part of user-mode accessible address space for protected memory means that we can use a base address of 0 for segments such as `ds`, `es` and `ss`, while the limit will be `0xb8000000`. Using a base address of zero provides maximum compatibility with existing code, as it avoids any need to adjust any pointers in memory, or when passing data to the kernel. Note that the code segment register is not limited in any way, thus permitting shadow code in the protected memory region to be executed.

By default, the OS maps the program stack at a high address, and often, this may overlap with the region that is needed to set aside. To resolve this conflict, SCInt relocates the stack at process start up time, and then sets aside the high memory region for memory segmentation protection. Next, SCInt initializes the segments. Note that segment descriptors are maintained in two tables in kernel space, LDT and GDT. Since GDT is a per-thread resource, we maintain the segment descriptors there. Specifically, the implementation uses index 7, which is currently unused, to set up protected thread-local storage that can be used by instrumentation, and index 8 to access unprotected memory. A system call policy is put in place to prevent any modifications to these entries. Protected thread local storage (also called "dedicated TLS") will be the only one that is permitted to access protected memory regions.

We now describe the instrumentation performed to enforce memory protection. First, all loads of data segment registers such as `ds`, `es` and `ss` are overridden to use the descriptor set up above for unprotected memory. Unfortunately, it is not possible to use a similar option for TLS segment registers such as `gs` and `fs`. This is because the current implementation of TLS on Linux requires access to memory with negative offsets, which will cause a segmentation violation unless the segment covers the entire 4GB address space. (This is the reason for the above-mentioned

[6]This limitation of the top 128MB is due to the fact that by default, the loader is mapped just below this 128MB address range. This range can be expanded by changing the default in the OS, or by changing the loader, but we have left these as future work.

problems with TLS support in existing memory segmentation implementations.) To deal with this, we use instrumentation to emulate the correct behavior of instructions that use these two segment registers. Of these two registers, `fs` is almost never used in 32-bit Linux, while `gs` experiences much more frequent use. Consequently, we use different strategies for emulation. In particular, we set aside `fs` so that it always points to the thread-local storage within the protected memory area. Any original uses of `fs` are always emulated. For `gs`, we emulate instructions that use offsets that cannot be statically verified to be within the unprotected region. For instance, an instruction `mov %eax, %gs:(%ebx)` will be replaced with the following snippet:

```
mov %edx, %fs : SPILL_DX
mov %fs : usr_gs_base, %edx
lea (%ebx, %edx, 1), %edx
mov %eax, (%edx)
mov %fs : SPILL_DX, %edx
```

Note that instructions in the program that load `gs` and `fs` will have to be sandboxed. Only the instrumentation code (and not the original program code) can load the `fs` register (or other segment registers) to point to memory range that includes the protected area.

In the x86-64 architecture, segmentation enforcement is not available. This lack of hardware support poses inconvenience to SFI implementations. An alternative approach is to transform all memory write instructions, this approach unfortunately incurs relatively high performance overhead. In avoid that, this work uses data randomization instead. The large address space of x86-64 architecture provides high entropy to hide the data. Moreover, SCInt ensures that memory accesses to protected memory could only be done through the unused TLS register (%`gs`)[7] with offset. Since segment base address is stored in kernel, any memory leak won't tell attacker the location of protected memory This gives attackers no clue to discover the data even they are able to read memory.

5.3.7 Support for dynamic code

Runtime changes to code may take place either due to the loading/unloading of libraries, or due to the use of just-in-time (JIT) compilation. The design described

[7] In x86-64, %gs register is not used by the glibc

so far already supports the first case of dynamic code. The second case, namely JIT code, poses some difficulties, and the following discussion explains below how compatibility with JIT can be obtained.

For best performance, many existing JIT compilers generate executable binary code in writable memory. This enables code to be updated very quickly. However, such an approach is inherently insecure under the threat model that this work considers, as an attacker that can corrupt memory can simply overwrite this code with her own code. For this reason, use of such JIT compilers is not advisable in this threat environment. Nevertheless, if compatibility with such JIT environment is desired, it can be supported by SCInt. Naturally, code pages generated by such a JIT compiler cannot be protected from modifications, but the design can continue to offer full protection for the rest of the code. This is achieved by marking JIT code region in a table and transforming JIT code at runtime. All control flows targeting JIT code will be redirected to its corresponding instrumented code where all indirect control transfers in JIT code will be checked.

A more secure approach for JIT support is one that avoids the use of writable code pages. Recent research work [131] have proposed a practical and efficient JIT code generation approach that eliminates writable code pages. This is achieved by sharing the memory that holds JIT code across two distinct address spaces (i.e., processes). Code generation happens in one of these address spaces, called software dynamic translator (SDT), where the page remains writable but not executable. JIT code execution happens in the second address space, regarded as the untrusted process, where the page is just readable and executable.

SCInt is compatible with this secure JIT code generator design as well. Each time new code is generated, it is instrumented by SCInt (and the underlying platform BinCFI). This instrumentation happens within the SDT process. The design of SCInt remains unchanged as far as the untrusted process process is concerned.

JIT code tends to change frequently, and the changes are typically small and localized. To obtain full benefits of JIT code, instrumentation efforts also needs to be localized, and performed in an incremental fashion. This requires some fine-tuning in the software dynamic translator to ensure SCInt works correcty. However, due to the fact that source code of secure dynamic code generator [131] is not available, the prototype does not implement incremental instrumentation for dynamic code.

5.4 Implementation

The state model and library loading policy are implemented by code instrumentation on the dynamic loader (`ld.so`). This work has instrumented code logic in `ld.so` that is used for code loading. In addition, this work has instrumented all the system calls that are located in loader, `libc.so` and `libpthread .so`. All the checks added by instrumentation will redirect control to a specially designed library loaded ahead of time. To ensure this library is loaded ahead of all other dependent libraries, this work uses environment variable `LD_PRELOAD`. Note that this library is independent of any modules including even the loader or libc. This ensures that the state model, library loading policy and system call monitor work immediately when the library is loaded.

To protect the special library from control flow and data attacks, the design of this work marks the entries in memory map table that corresponds to location of the special library as empty. Thus any subverted code pointers targeting the library will crash the program once used. To further protect this library from data attacks, the checks only use protected memory mentioned in Section 5.3.6. Finally, the special library uses its own stack in protected memory.

5.5 Evaluation

We implemented the application system SCInt on top of PSI. The test environment is Ubuntu 12.04 LTS 32bit, with Intel i5 CPU and 4GB memory.

5.5.1 Micro-benchmark for program loading

Runtime overheads arise chiefly from the following: (1) larger size of the binary to be loaded, (2) checks performed in the context of the state model, (3) checking of library policies. Almost all of these overheads occur at the time of loading a library, so this work focuses on the increase in runtime for performing library loads. To measure this overhead, the author wrote a small program that loads a number of automatically generated libraries. This work measured the runtime needed to load the original version and compared it with that for loading the

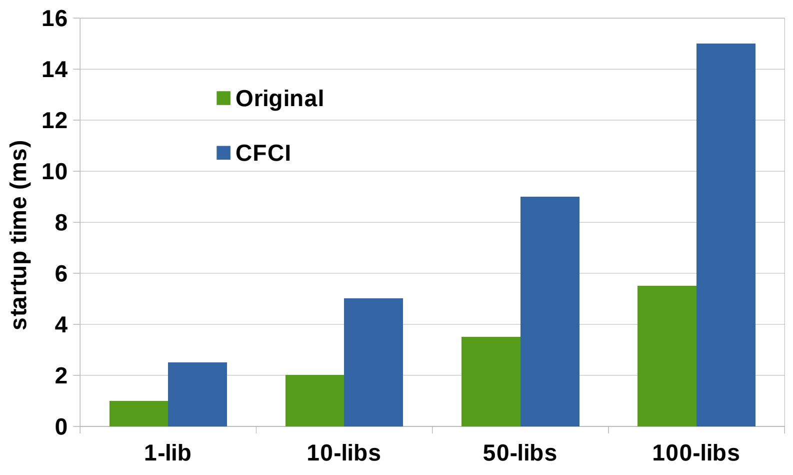

Figure 5.6: Micro Benchmark Evaluation of SCInt

transformed versions.[8] Figure 5.6 demonstrates that the overhead grows linearly with the number of libraries loaded. On average, the overhead of SCInt on a library load is 150%.

While this overhead may seem considerable, it should be kept in mind that this is a microbenchmark, and the actual overheads on loading and running entire application is much smaller.

5.5.2 CPU intensive benchmark

Since SCInt does not introduce significant additional operations at runtime, one would expect that it does not introduce significant overhead. Specifically, the only additional overhead of SCInt is due to policies on operations for memory mapping or protection, and file-related operations such as open, close and read. Since these operations are relatively infrequent in comparison with the number of instructions executed, and since the policy checks themselves are quite simple, it is expected that the overhead should be small.

To validate this assessment, the author evaluated SCInt with SPEC 2006 using reference dataset. The overall overhead of SCInt observed is 14.37%, which has 0.17% higher than that obtained with just BinCFI. The source of the overhead comes from system call interception. Figure 5.7 shows the details of the experiment.

[8] Note that this work transformed all libraries for when testing transformed binary.

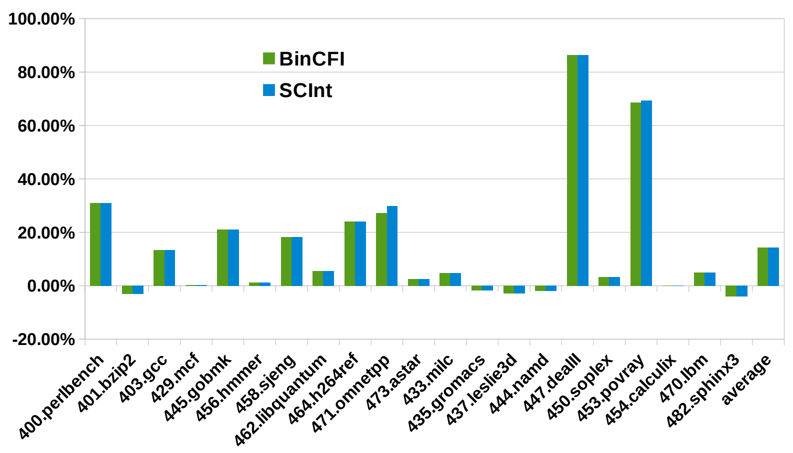

FIGURE 5.7: Performance of SCInt on SPEC2006

Program Name	Base load time	BinCFI overhead (over base, %)	Added SCInt overhead (over base, %)	# of loaded libraries
vim	0.140	34	8	96
evince	0.336	222	8	103
lyx	0.484	89	10	152
lynx	0.052	38	2	15
wireshark	0.684	224	18	114
nautilus	0.080	900	16	178
acroread	0.972	280	7	82

FIGURE 5.8: Startup Overhead on Real World Programs

5.5.3 Commonly used Linux applications

In addition to the above tests involving the SPEC benchmarks, this work also selected several real world applications widely used on Linux. Since only code allocation and deallocation will generate performance overhead, the focus of the evaluation turns to program startup when large number of modules get loaded. The results are shown in Figure 5.8, where the base load time is in seconds.

From Figure 5.8, it is clear that BinCFI has observable startup overhead, 147% on average, while the overhead generated by SCInt on top of BinCFI is small, 8% on average. The loading phase in BinCFI is not optimized as their focus was on the execution phase performance rather than loading performance. The additional overhead of SCInt ranges from a low of 2% for programs that load just a few libraries, to a high of 18% for programs that load a large number of libraries. Note that many of these programs load more than 100 shared libraries each, and hence the average overhead reported is likely to be a conservative estimate.

5.5.4 Running with dynamic code

To demonstrate compatibility with dynamic code, the author used LibJIT [16], an actively maintained JIT engine similar to the LLVM backend. To make LibJIT compatible with SCInt, the author forced LibJIT to generate only non-writable code. To evaluate the performance overhead for LibJIT, the author reuses an open source benchmark tool [42]. The benchmark tool is a simple program that computes the greatest common denominator (GCD). It generates the GCD function code dynamically at runtime. LibJIT allows dynamic functions to be invoked directly (LibJIT-fast). The evaluation shows that SCInt overhead is 14.6%. This overhead includes 10% overhead incurred for runtime code transformation.

Note that the prototype does not currently support JIT code update. This limitation is reasonable because (a) if JIT code update is allowed, the whole security property enforced by SCInt may be invalidated unless a safe update of dynamic code design is applied.[9] and (b) there will be no visible difference on performance if a safe JIT code generator is used, since code update requires the same amount of time by the helper process and the only overhead is the IPC communication between target application and helper process. Because of that, the evaluation does not includes overhead for JIT code update.

5.5.5 Effectiveness evaluation

Although CFI bypass techniques are emerging as shown in recent research work [62, 74], real-world exploits have not been designed with CFI in mind. Therefore, running those exploits won't show the benefits of SCInt. For this reason, the effectiveness evaluation uses a combination of manual analysis based on studying the relevant CVE reports, and proof-of-concept exploits that is created by the author. This evaluation is summarized in Figure 5.9.

According to the threat model in Section 5.2, the author classifies all attacks into direct attacks (`direct`), loader data corruption attacks (`ldr.data`), and loader code reuse attacks (`ldr.cr`). From Table 5.9, it is clear that all types of attacks are defeated. In particular, system call policy prevents all direct attacks that try to manipulate code permission or launch malicious binaries. For loader subversion attacks such as search path corruption, library loading policy properly defeats all

[9]However, safe JIT code generator is currently not available. The work proposed by recent research [131] is based on Strata. Its source code in unavailable.

Attack	Type	Detail	Blocked?	Reason of Rejection
ROP-CVE-2014-1776	direct	alloc executable area and jump	Y	syscall policy
ROP-CVE-2014-1761	direct	launch malicious executable	Y	syscall policy
ROP-CVE-2014-0497	direct	change page permission, download exe and launch it	Y	syscall policy
ROP-CVE-2012-1875	direct	make heap executable	Y	syscall policy
CVE-2013-3906	direct	alloc executable data and jump	Y	syscall policy
CVE-2013-0977	ldr.data	malformed binary with overlapping segments	Y	state model
CVE-2014-1273	ldr.data	malformed binary with text relocation	Y	state model
CVE-2010-3847	ldr.data	library hijacking using $ORIGIN in LD_AUDIT	Y	library loading policy
CVE-2010-3856	ldr.data	library hijacking on LD_AUDIT	Y	library loading policy
CVE-2011-1658	ldr.data	library hijacking using $ORIGIN in RPATH	Y	library loading policy
CVE-2011-0570	ldr.data	Untrusted search path vulnerability in Adobe Reader	Y	library loading policy
ROP-CVE-2013-3906	ldr.cr	load malicious library	Y	library loading policy
PoC: attack-lder-1	ldr.cr	code injection via corrupted reloc table and sym table	Y	state model
PoC: attack-lder-2	ldr.cr	code injection via making stack executable	Y	syscall policy
PoC: attack-lder-3	ldr.cr	loading malicious library by calling _d_lopen	Y	library loading policy

FIGURE 5.9: Effectiveness Evaluation of SCInt

```
Policy for Adobe Reader:

ALLOW    libc.so.6        /lib/i386-linux-gnu/
REJECT   libc.so.6        *
ALLOW    libpthread.so.0 /lib/i386-linux-gnu/
REJECT   libpthread.so.0 *
ALLOW    libselinux.so.1 /lib/i386-linux-gnu/
REJECT   libselinux.so.1 *
ALLOW    *            /lib/i386-linux-gnu/
                      /usr/lib/*
                      /opt/Adobe/Reader9/Reader/
                      intellinux/lib
                      /usr/lib/i386-linux-gnu/*
                      /usr/lib
REJECT   *            *
```

FIGURE 5.10: Library Loading Policy for Adobe Reader

unallowed module being loaded from unintended path. Moreover, more advanced attacks such as code reuse attacks targeting the loader are stopped by the code loading state model. To provide more insight in this regard, this work considers the following case studies.

Case study: library policy for Adobe Reader As described in Section 5.2, an attacker may attempt to load a malicious library by specifying it by name, or by corrupting the load path ("library hijacking"). SCInt blocks these attacks using policies to limit the load path, as well as the specific libraries that may be loaded by an application. To illustrate these policies, consider Figure 5.10 which shows the policy for Adobe Reader, a favorite target for library loading attacks [7–9].

In addition, the example also illustrates the flexibility provided in the policy language. It is possible to allow or deny loads of specific libraries, or permission can be granted based on the directory from which a library is loaded. Moreover, policies can be stricter for some libraries. Figure 5.10 uses a stricter policy for low-level libraries `libc.so`, `libpthread.so` and `libselinux.so`, forcing them to be loaded from a specific file in a specific directory.

Case study: library policy for static binary It is a common misconception that static binaries won't have any ability to load libraries. Experiments prove otherwise. The author wrote a small, statically-linked program that contains a traditional buffer overflow vulnerability, and crafted an exploit for this vulnerability. Instead of calling `dlopen`, a function that is unavailable in statically linked binaries, the exploit redirected control to `_dl_open`, an internal function statically linked from loader. This function was passed the name of a library in /`tmp`. The exploit resulted in a successful load of this library. Along with the library, dependent libraries such as `libc.so` and `ld.so` were also loaded! Finally, the initialization function of the malicious library was executed.

The default policy for library loading stopped this attack, as it prevents loads of libraries outside standard directories for libraries.

Case study: Text relocation attack In order to evaluate code-reuse and data corruption attacks on the loader, implemented a proof-of-concept exploit that leverages text relocations.

The exploit code is implemented as a piece of native code. This helps us avoid the complications of crafting valid ROP attacks in the presence of CFI. Full details of the attack can be found in Appendix 5.2.3, but it suffices for the purposes of this paper to outline the general idea, which is to corrupt the loader's data structures and launch code injection by reusing code patching ability in the loader. The experiments shows that this exploit is very robust and works on any dynamically linked ELF programs.

The author then attempted the exploit in the presence of SCInt. The attack was detected and blocked by the policies shown in Figure 5.4.

Case study: Making stack executable Similar to relocation attack, the author wrote a simple proof-of-concept exploit that makes the stack executable, and then jumps to the stack. In glibc 2.15, the loader function for making the stack executable makes two sanity checks. The first one checks that the caller's address is within the loader, while the second one checks the consistency of `__libc_stack_end`. Bypassing the first check is easy — simply using a return address in loader would suffice; when the function is finished, it will go back to a gadget inside the loader and can be arranged to jump back to the call site. Bypassing the second check requires the attacker to corrupt a global variable in `ld.so`, which is easy once ASLR is bypassed. In summary, the exploit code bypassed these two checks and made the stack executable.

When run in the presence of SCInt, this exploit is blocked by the policies shown in Figure 5.4. Specifically, the attempt made by the exploit to make an empty region (ES) executable was denied by this policy.

Defeating this attack is important despite the fact that BinCFI alone could already defeat it. This is because, other underlying CFI implementation may rely on DEP. For instance, Abadi CFI [28] requires stack to be non-executable to prevent jump-to-stack attack that has valid ID in payload.

5.6 Summary

In this chapter, we presented an effective countermeasure against the threat of ROP attacks. The approach is based on the observation that the goal of real-world code-reuse attacks is to disable DEP and launch a native code injection attack. The defense combines coarse-grained control-flow integrity with a comprehensive defense against native code injection in order to defeat these attacks. This approach tracks the code loading process through every step, and ensures that code integrity is preserved at every step. Consequently, it can ensure a strong property that only authorized (native) code can ever be executed by any process protected by this system, SCInt. A key benefit of this approach is that its security relies on a relatively simple state model for loading, and a few simple system call policies. It is fully compatible with existing applications as well libraries and loaders, and does not require any modifications at all. SCInt introduces almost no additional overheads at runtime over CFI, making it a promising candidate for deployment.

Chapter 6

Related Work

6.1 Static binary disassembly

Many efforts in exploit protection and hardening of binaries have been made using static binary instrumentation. However, supporting large and complex COTS binaries has proved elusive due to the twin challenges of accurate disassembly, and safe instrumentation in the presence of embedded code pointers within binaries. To overcome the disassembly problem, most previous binary instrumentation efforts have relied on a cooperative compiler [146], or worked on assembly code [28, 95, 108], or binary code containing symbol table [87, 119], debugging [29, 67], or relocation information [151]. CCFIR [152] uses the runtime rebasing information in Microsoft dynamic-link library (DLL). Due to ASLR, this information is mandatory even in stripped DLLs.

SecondWrite [33] uses static binary instrumentation to harden binaries. It disassembles the binaries speculatively, lifts them up into LLVM internal representation (IR) and uses LLVM backend to regenerate binaries. They have shown that their platform handles moderately large applications [32]. Their focus is on powerful analysis and optimization techniques that provide excellent performance. However, support of reliable disassembly for shared libraries or low level library code such as glibc remains as an open problem.

Several earlier works such as BIRD [99] and Dyninst [53] used a combination of static analysis, compiler idioms and heuristics to disassemble binaries. BIRD further improves results by incorporating a runtime disassembly component. However, use of heuristics means that there could be disassembly errors, and these can lead to instrumentation subversion, e.g., by jumping to the middle of an instruction. Other research efforts based of IDA-pro such as in-place-randomization (IPR) [109] share the same issue because the purpose of IDA-pro is to maximize the code disassembly in the context of use for malware analysis or other analysis by human analyst. Zero disassembly error is not the goal. On the other hand, both BIRD and IPR target binaries on Windows, where disassembly can be more complex.

REINS [141] targets sandboxing of untrusted COTS executables on Windows. REINS uses IDA-pro as its baseline disassembler and uses IDApython to correct disassembly errors and harvest high level information such as function boundaries and etc for static analysis. Similar to PSI, REINS can also ensure that sandboxed code can never escape instrumentation (except to invoke certain trusted functions), but unlike techniques in PSI, their approach does not target instrumentation of all libraries used by the application. Moreover, their evaluation does not consider as many or as large applications as PSI.

In contrast, our platform PSI provides a robust binary disassembly algorithm that could handle large and complex COTS binaries without disassembly errors.

6.2 Static binary analysis

Previous research demonstrates that static binary analysis is essential for securing COTS binaries [76, 141, 152, 153]. Bao et. al. [39] demonstrate that higher level information such as function boundaries could be discovered with high accuracy. They use binary code pattern as heuristics to identify function boundaries. Furthermore, several efforts [66, 81] have been made to recover variable and type information from COTS binary code.

The focus of the static analysis in PSI and others mentioned are to use a conservative analysis that for securing code. Binary analysis could also be speculatively performed to do malware analysis and exploit generation. Binary analysis platform (BAP) [52] is a platform that targets COTS binaries. They have developed interesting applications such as automatic exploit generation [35]. BitBlaze [133] also targeted powerful static and dynamic analysis of binary code, including malware. Similarly, Codesurfer [38] targets malware analysis. In particular, it uses a static analysis algorithm call value-set analysis (VSA) to recover intermediate representations that represent high level semantics of target binaries.

6.3 Binary instrumentation

Dynamic binary instrumentation Dynamic binary instrumentation systems can be categorized into: (a) whole system DBI and (b) application-level DBI. Whole system DBI tools take care of every instruction execution and emulate some of those that require privileges, while application level DBI only handles user level instructions of one target application process. Tools such as QEMU [41] and Bochs [26] and others [54, 147, 150] fall into whole system DBI. We do not focus on the discussion of whole system DBI since it is less relevant to our work.

Application-level DBI tools only handle instructions in the user level. Valgrind [102] allows programs compiled in one architecture to run in another architecture. It works by translating every instruction into its internal representation and thus suffers from large runtime overhead. Most of application level DBI tools work on the same architecture as the target programs. Therefore, instruction translation in these tools can be avoided. Tools such as DynamoRIO [51], Pin [90], Strata [122], StarDBT [49] work in the user level by injecting themselves into address space of the target program.

Since application-level DBI works in the same address space of the target application, one of the key requirements is that the impact of the tool should be kept invisible to target program. DBI tools achieve this by using their own stack and heap and keep away from memory and system resource used by target programs.

Comparative analysis of SBI and DBI Although application-level DBI tools are relatively more efficient, the cost of using code cache is significant. First of all, usage of code cache requires a “cold start” of each program execution since code cache is built at runtime on each execution. This poses a substantial overhead at program startup, limiting its usage in certain scenarios. Another practical limitation of DBI is the security compromise that comes with writable and executable code cache. For performance reasons, code cache is configured to be writable and executable to cope with frequent updates. This gives attackers an opportunity to corrupt code cache and bypass instrumentation. Song et. al. [132] present their prototype that retrofits existing DBI system Strata by using non-writable code cache. They use double processes where code cache is shared across two processes, but is writable only in the helper process. Modern DBI tools are not yet to adopt such techniques to protect code cache.

Static binary instrumentation avoids using code cache in principle because instrumentation is performed ahead of time. In addition, the performance on program startup will be better.

6.4 Control flow integrity

Control-flow integrity (CFI) was introduced by Abadi et al [28]. The basic idea was to use a static analysis to compute a control-flow graph, and enforce it at runtime. Enforcement was based on matching constants (called IDs) between the source and target of each ICF transfer. However, due to difficulties in performing accurate points-to analysis, and because of so-called destination equivalence problem, their implementation resorts to coarse granularity enforcement, wherein any indirect call is permitted to target any function whose address is taken. Li and Wang et. al. [89] and HyperSafe [139] implement a compiler based CFI that uses a similar policy for coarse-grained CFI. While they can also support finer-granularity CFI, this requires runtime profiling to compute possible targets of indirect calls, and can hence be prone to false positives.

Control-flow locking (CFL) [47] improves significantly on the performance of original CFI [28], while simultaneously tightening the policy, especially for returns. But this tighter policy poses challenges in the presence of indirect tail calls. Another difference between their work and ours is that they operate on assembly code generated by the compiler, whereas our work targets binaries.

MoCFI [59] presents a design and implementation of CFI for mobile platforms. The mobile environment presents a unique set of challenges, including an instruction set that does not have explicit returns, a closed platform (iOS), and so on. An important characteristic of their approach is that they aggressively prune possible targets of each ICF transfer. While this can provide better protection, it leads to false positives in some cases (e.g., when large jump tables are involved). In contrast, we emphasize handling of large binaries, including shared libraries, that are not handled by their approach. We discussed how this requirement dictates the use of coarser granularity CFI in PSI.

CCFIR [152], like the work presented in this dissertation, targets binaries. The main insight in their work is that most binaries on Windows support ASLR, which requires relocation information to be included in the binary. They leverage this information for accurate disassembly and static instrumentation. Moreover, since relocation information effectively identifies all code pointers, they can avoid runtime address translation, which enables them to achieve better performance. The flip side of this performance improvement is that the technique can't be used on most UNIX systems, as UNIX binaries rarely contain the requisite relocations.

CFI has been used as the basis for untrusted code sandboxing. PittSFIeld [95] implements SFI on top of instruction bundling, a weaker CFI model. XFI [67] presents techniques that are based on CFI and SFI to confine untrusted code in shared-memory environments. Zeng et al [149] improve the performance of SFI using CFI and static analysis. Native Client [146] is aimed at running native binaries securely in a browser context, and relies on instruction bundling. PittSFIeld, Native Client, and many other works [30, 34, 82, 84, 89, 113, 123, 139] that enforce CFI rely on compiler-provided information and even hardware support. In contrast, BinCFI operates on COTS binaries without support from compiler, OS or hardware. In addition, it could be applied to all binaries including executable and their dependent libraries.

Recent research has shown that coarse grained CFI approaches such as BinCFI, CCFIR allow for enough CFI-permitted gadgets to construct functional ROP payloads [55, 62, 72, 73]. Fine-grained CFI [60, 105, 106, 136] can mitigate some of these attacks, but it is unlikely to provide a fool-proof defense. Moreover, as discussed earlier, finer-grained CFI enforcement comes with compatibility problems.

Instead of improving granularity, Opaque CFI (O-CFI) [97] adopts randomization to hide the applied policy. In particular, it constrains the target range of indirect control transfer instructions to a destination set expressible as a bounds check. The bounds lookup table is protected using memory segmentation and information hiding. There are several limitations in O-CFI: (a) Deriving a precise set of allowed targets, however, is challenging even if source code is available for static analysis [61], (b) randomized bounds are still located in the target code address space and thus it suffers from blind ROP attacks and (c) O-CFI still leaves code readable, and thus it is unclear how difficult it is for attacker reverse engineer their policy back.

In contrast, our instrumentation application, SECRET, does not suffer any for these limitations. In particular it uses code pointer remapping to overcome the first two limitations of O-CFI and uses code space isolation to overcome the third limitation.

6.5 Code randomization

Code randomization can be applied by leveraging the randomness of instruction semantics [?] or machine code layout [45]. The former approaches including ISR [?] aim to defeat native code injection attacks. However, they cannot defeat code reuse attacks and suffer from high overhead. Code layout randomization offers an additional layer of protection over ASLR and its shortcomings [127]. They can be applied at varying levels of granularity, e.g., at the function [44, 45, 83], basic block [140], or instruction level [64, 75, 109]. The deployment time spans across compile time [76] , load time [140]. G-Free [108] is a compiler-level instruction transformation technique against ROP. The applied transformations remove gadgets within unintended code, but the adopted strategy for intended gadgets is weak.

None of the code randomization defenses defeat code reuse attacks that harvesting code or code pointers. Thus, attacks such as JIT ROP [130] bypass them in general.

6.6 Code disclosure attacks and countermeasures

Control flow integrity and code randomization protections are challenged by advanced attacks that leverage memory disclosure vulnerabilities to dynamically construct ROP payloads [46, 130]. JIT-ROP [130] repeatedly uses a memory disclosure to read executable memory and chain discovered gadgets to launch a ROP attack. Blind ROP [46] leverages a stack buffer overflow in forking servers to repeatedly overwrite the stack until the `write` function is located, which then is used to leak executable process memory to the client. Under certain circumstances, even if a memory disclosure bug is not available, gadget locations can be inferred through side channels [124]. The same technique can be used to infer other critical data, as was recently demonstrated in a bypass attack against CPI in certain configuration [68].

As a step towards protecting against memory disclosure vulnerabilities, recent research efforts attempt to make code executable but not readable, by relying on page table manipulation [36], split TLBs [71]. However, neither of these could defeat JIT ROP attacks that havest code pointers only. Recent efforts on code randomization have shown that JIT-ROP [130] attacks can be mitigated. Oxymoron [37] applies fine grained code randomization that is compatible with code sharing. It offers some resistance to memory leakage attacks by replacing direct branches with indirect branches. This makes it harder for attackers to harvesting code pages by following control flows.

The authors of Isomeron [61] propose a JIT ROP attack that bypasses Oxymoron by harvesting code pointers in the user memory such as the stack. Those code pointers are then dereferenced to read code. In the work of Isomeron [61], they further propose a JIT ROP defense that combines execution path and code randomization. The purpose of the approach is to reduce the probability of correctly predicting the runtime addresses of gadgets. Isomeron maintains two copies of a program's code in the same address space, and at runtime program execution may switch from one version to another or stay unchanged. Switches are decided on `call` and `ret` instructions. Decision information is hidden within the large address space to prevent attackers from reading it. A ROP payload assembled during a JIT-ROP attack is likely to divert control to the wrong code version and suffer from unpredictable gadgets. While Isomeron provides additional strong randomization, its feasibility relies on the pairing of `call` and `ret` instructions, and thus it shares the same compatibility issues with well-known techniques such as shadow stacks [63]. In comparison, SECRET does not suffer from compatibility issue since return addresses could be used for purposes other than function return. In addition, SECRET achieves the same level of strength as Isomeron since it defeats both JIT ROP attacks that read code directly and indirectly through the usage of pointers while maintains compatibility with real world programs.

Readactor [58]is a very recent effort to defeat JIT ROP attacks. Readactor uses extended page table (EPT) to ensure that all code pages are not readable to prevent JIT ROP attacks that directly read code. In addition, it changes all code pointers pointing to "proxy" pages that contains trampoline code stubs. By doing so, JIT ROP attacks that harvest code pointers are defeated because leaked code pointers all point to "proxy" pages that leak no information to attackers. Since no code pointers are leaked pointing to code pages, Readactor could use a simple randomization to make sure attackers have no idea of hidden code layout.

For defeating JIT ROP attacks that aim at code reading, SECRET and Readactor achieve the same effect. SECRET defeats JIT ROP attacks by hiding the code and ensure that no pointer pointing to it, while Readactor leverages EPT and compiler support respectively. For hiding code pointers, readactor randomizes all code pointers since it operates on source code, while SECRET can safely change the majority of code pointers including return addresses and exported functions.[1] The strength of SECRET over Readactor is that it defeats Blind ROP attacks. However, it may suffers from call oriented programming attacks.

6.7 Code injection prevention

MIP [104] and MCFI [105] incorporate some code integrity features other than ensuring control flow integrity. Their policies are implemented using checks on `mmap` and `mprotect`. However, the policy only aims to ensure existing code is never writable and executable. Moreover, the policy does not constrains the code loading/patching privileges of dynamic loader. Attackers can divert control flow to dynamic loader code and leverage its privileges to bypass their policy.

Nanda et. al. [100] proposes an approach to detect foreign code in Windows. Similar to our approach, they enforces a code loading policy to prevent malicious library loading. In addition, they prevent malicious operations that change code permission. However, their approach suffers from several limitations: (a) text relocation is not protected and potentially suffer from the attack mentioned in Section 5.2.3,[2] and (b) their work is built on top of BIRD [99], a dynamic binary translator that uses JIT code, which by itself suffers from low level code corruption attack.

Realizing the importance of preventing abuse of code loading privileges, Payers et.al. [112] developed TRuE, a system that replaces the standard loader with their secure version. The need to replace the system loader poses challenges for

[1]SECRET does not change all code pointers. This is due to the general limitation of COTS binary analysis that identifying code pointers and integers in COTS software is undecidable in general.

[2]Note that all Windows binaries on x86 requires text relocations to support ASLR

real-world deployment, as OS vendors are reluctant to change core platform components. There is a strong interdependence between the loader and glibc, and as a result, a replacement of the loader also requires changes to the glibc package. Another difficulty with their approach is that the secure loader achieves security in part by restricting the functionality of the loader. Our approach avoids these drawbacks by permitting continued use of the standard loader. Security is achieved by a small and independent policy enforcement layer that operates outside the loader, and simply checks the security-relevant operations made by the loader.

Writing secure loaders is a very difficult task, as demonstrated by the numerous vulnerabilities reported in production loaders [4–6, 10, 13–15]. Thus, trusting the complete loader code for code integrity leads to a large trusted computing base (TCB). In contrast, SCInt implementation is a very small reference monitor on top of existing system. The code size is no more than 300 LoC including C and x86 assembly code. In comparison, a dynamic loader is 28K LoC.

Some efforts have been made toward code injection defense in OS level. iOS and OS X are using XNU kernel which leverages code signing to secure code loading. In addition, the kernel enforces that all code pages that are signed/verified cannot be modified at runtime. This approach makes their genuine systems safe from code injection. However, the continuing success of jailbreaks proves that code signing could be bypassed by using kernel vulnerabilities. In all these scenarios, SCInt is the complement approach for code injection attacks.

PaX team provides security patches for Linux kernel. Their project `grsecurity` provides several kernel security enhancements. One of them is the restricted `mprotect` [135]. The basic idea is similar to SCInt that provides code integrity. Their enhanced kernel will check the parameters of `mprotect` and deny any request to make code pages writable except once for those ELF images that contain text relocations. The limitation of restricted `mprotect` is that it does not ensure code integrity protection when an ELF image has text relocations. By contrast, SCInt is able to achieve that.

In Windows, binary loading primitive is implemented by the kernel. Therefore, applications in Windows do not suffer from attacks at loading time. This makes it easier for SCInt. However, the vulnerability of windows binary loading lies in the heavy usage of text relocations due to dynamic linking and ASLR. Text relocations could be crafted by attackers to modify existing code as mentioned in Section 5.2.3.. SCInt could prevent those attacks, since none of the original code could be executable.

Compared with code injection defense on OS level, SCInt shines in these aspects: (a) it can be securely implemented entirely at the user level, (b) its security relies on a small piece of code (state model), and (c) it can handle complications such as text relocation.

6.8 Defenses against memory corruption

Several comprehensive bounds-checking techniques have been proposed for C/C++ programs [31, 80, 98]. They operate by maintaining additional metadata about pointers, and checking this metadata to detect attacks. Unfortunately, they impose significant overheads, and moreover, lead to compatibility problems with large and complex applications. Code pointer integrity (CPI) [86] significantly reduces both the compatibility and performance problems by applying bounds-checking selectively to those pointers whose corruption can lead to control-flow hijacks. However, since the underlying techniques are similar to SoftBounds, compatibility problems cannot be totally eliminated.

Source-code based techniques such as CPI [86], Baggy [31], and MCFI [105] can only be applied to C/C++ source code. In particular, low-level code that relies on inline assembly will not be protected. Similarly, code compiled from higher level languages other than C/C++ are left out. Finally, any code that is available only in binary form cannot be protected. Unfortunately, even if a single module is not compiled for bounds-checking, no security guarantees can be provided for any code. Thus, the “weakest link” can potentially derail the entire the application. It is in this context that purely binary-based approaches such as SCInt shine: *SCInt’s protection extends to all code, regardless of the language in which it is written, or the compiler used to compile it.*

Chapter 7

Conclusions and Future Work

Securing COTS software is a challenging task. In this dissertation, we showed that static binary instrumentation can be effectively used to work with complex COTS binaries and secure their execution. We hence developed our system PSI. By using this instrumentation system, we next explored defenses against advanced code reuse attacks and developed a complex application system called SECRET. For those ROP attacks that in principle still exist despite of SECRET, we developed a systematic approach called Strong Code Integrity (SCInt) to further mitigate them and ensure that they cannot launch code injection attacks.

Our platform, called PSI, uses a practical disassembly algorithm as well as a conservative static analysis to recover the control flow graph of a target binary. Then it uses instrumentation with address translation to support arbitrary instrumentation. PSI incorporates BinCFI, a practical CFI policy that works for complex and large COTS binaries. The instrumentation developed is modular and complete since it can be applied to executables and all their dependent libraries. We have made an extensive evaluation on both the SPEC 2006 benchmarks as well as several commonly used Linux applications. We also compared the performance of our system compared with that of state-of-art DBI systems such as DynamoRIO and Pin. The result shows that on the SPEC 2006 benchmarks, PSI achieves comparable performance with DBI tools, while on commonly used application benchmarks, PSI is 7 and 13 times faster.

We presented two significant security applications on top of PSI: SECRET and SCInt. In particular, SECRET aims to defeat advanced code reuse attacks such as ROP. SECRET is supported by two techniques: code pointer remapping and code space isolation. Code pointer remapping is used to defeat code reuse attacks that leverage prior knowledge of binary contents, while code space isolation defeats code reuse attacks that harvesting code pages or code pointers at runtime.

The purpose of SCInt is to ensure executable code integrity even if code reuse attack are not fully defeated. We proposed a systematic approach against code injection attacks to make sure that (a) only permitted module can be loaded and executed in memory and (b) only legitimate instructions in these modules can be executed. Unlike Code signing which only ensures code integrity at load time, SCInt ensures code integrity for the whole runtime. It effectively mitigates advanced code reuse attacks and naturally complements all existing code reuse defenses.

Together, PSI, SECRET and SCInt present an integrated framework for securing COTS binary execution against advanced stealthy attacks such as return oriented programming and code injection. Meanwhile, they also suggest many interesting research problems. Below, we briefly conclude this dissertation with an open problem in the research direction:

Protecting COTS Just-in-time compiler: Just-in-time (JIT) compiler is a special piece of software that generates code dynamically. Secure execution of dynamic code is an important missing part and remains an open research question. Despite the fact that there are several efforts having been made to protect dynamic code. None of them could work on COTS JIT compiler, i.e. they all require modifications on source code. This would pose limitation on protecting software such as Internet explorer, adobe flash player and etc.

Bibliography

[1] bincfi v1.0. http://www.seclab.cs.sunysb.edu/seclab/bincfi/.

[2] CVE-2000-0854: Earliest side-loading attack.

[3] CVE-2007-3508: Integer overflow in loader. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3508.

[4] CVE-2010-0830: Integer signedness error in loader. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0830.

[5] CVE-2010-3847: privilege escalation in loader with $origin for the ld_audit environment variable. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3847.

[6] CVE-2010-3856: privilege escalation in loader with the ld_audit environment. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3856.

[7] CVE-2011-0562: Untrusted search path vulnerability in adobe reader.

[8] CVE-2011-0570: Untrusted search path vulnerability in adobe reader. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0570.

[9] CVE-2011-0588: Untrusted search path vulnerability in adobe reader. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0588.

[10] CVE-2011-1658: privilege escalation in loader with $origin in rpath. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1658.

[11] CVE-2011-2398: privilege escalation in loader. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-2398.

[12] CVE-2012-0158: Side loading attack via microsoft office. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0158.

[13] CVE-2013-0977: overlapping segments. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0977.

[14] CVE-2013-3906: arbitrary code execution via a crafted tiff image. http://www.fireeye.com/blog/technical/cyber-exploits/2013/11/the-dual-use-exploit-cve-2013-3906-used-in-both-targeted-attacks-and-crimeware-campaigns.html.

[15] CVE-2014-1273: text relocation. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1273.

[16] LibJIT. https://code.google.com/p/libjit-linear-scan-register-allocator/.

[17] Lmbench tool for performance analysis. http://lmbench.sourceforge.net/.

[18] ROP attack exploiting CVE-2013-3906 to load a malicious library. https://www.fireeye.com/blog/threat-research/2013/11/the-dual-use-exploit-cve-2013-3906-used-in-both-targeted-attacks-and-crimeware-campaigns.html.

[19] ROP attack exploiting CVE-2014-0497 using VirtualProtect. http://blogs.technet.com/b/mmpc/archive/2014/02/17/a-journey-to-cve-2014-0497-exploit.aspx.

[20] ROP attack exploiting CVE-2014-1761 to execute malicious PE executable. http://blogs.mcafee.com/mcafee-labs/close-look-rtf-zero-day-attack-cve-2014-1761-shows-sophistication-attackers.

[21] ROP attack exploiting CVE-2014-1776 using VirtualAlloc. http://blog.fortinet.com/post/a-technical-analysis-of-cve-2014-1776.

[22] ROP attack exploiting CVE-2014-8439 to download malicious DLL. http://blogs.technet.com/b/mmpc/archive/2014/12/02/an-interesting-case-of-the-cve-2014-8439-exploit.aspx.

[23] Ropgadget. http://shell-storm.org/project/ROPgadget.

[24] Source code instrumentation overview at ibm website. http://www-01.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.doc/topics/cinstruovw.html.

[25] Winsxs: Side-by-side assembly. http://en.wikipedia.org/wiki/Side-by-side_assembly.

[26] bochs: The open source ia-32 emulation project, 2001. http://bochs.sourceforge.net/.

[27] MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit, 2013. http://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/.

[28] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[29] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM TISSEC*, 2009.

[30] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Security and Privacy (Oakland)*, 2008.

[31] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security*, 2009.

[32] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *EuroSys*, 2013.

[33] K. Anand, M. Smithson, A. Kotha, K. Elwazeer, and R. Barua. Decompilation to compiler high IR in a binary rewriter. Technical report, University of Maryland, 2010. http://www.ece.umd.edu/barua/high-IR-technical-report10.pdf.

[34] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI*, 2011.

[35] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: automatic exploit generation. In *NDSS*, 2011.

[36] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *CCS*, 2014.

[37] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security*, 2014.

[38] E. G. Balakrishnan, G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86—a platform for analyzing x86. In *LNCS*, 2005.

[39] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *USENIX Security*, 2014.

[40] J. Barker, Elaine; Kelsey. Recommendation for random number generation using deterministic random bit generators, 2013.

[41] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX ATC*, 2005.

[42] E. Bendersky. LibJIT Samples. https://github.com/eliben/libjit-samples, 2013.

[43] J. Bennett, Y. Lin, and T. Haq. The Number of the Beast, 2013. https://www.fireeye.com/blog/threat-research/2013/02/the-number-of-the-beast.html.

[44] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *USENIX Security*, 2003.

[45] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security*, 2005.

[46] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Security and Privacy*, 2014.

[47] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *ACSAC*, 2011.

[48] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS*, 2011.

[49] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO*, 2006.

[50] D. Bruening and Q. Zhao. Building dynamic instrumentation toolswithdynamorio. *Tutorial at CGO*, 2011.

[51] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, MIT, 2004.

[52] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. BAP: a binary analysis platform. In *CAV*, 2011.

[53] B. Buck and J. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 2000.

[54] P. P. Bungale and C.-K. Luk. Pinos: A programmable framework for whole-system dynamic instrumentation. In *VEE*, 2007.

[55] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses.

[56] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *CCS*, 2010.

[57] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *the 5th International Conference on Information Systems Security (ICISS)*, 2009.

[58] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Security and Privacy*, 2015.

[59] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A. reza Sadeghi. MoCFI: a framework to mitigate control-flow attacks on smartphones. In *NDSS*, 2012.

[60] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*, 2014.

[61] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

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

[63] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS*, 2011.

[64] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *ASIACCS*, 2013.

[65] S. Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63.

[66] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *PLDI*, 2013.

[67] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *OSDI*, 2006.

[68] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Security and Privacy*, 2015.

[69] A. D. Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. How the elf ruined christmas. In *USENIX Security*, 2015.

[70] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC*, 2008.

[71] J. Gionta, W. Enck, and P. Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 325–336, 2015.

[72] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Security and Privacy*, 2014.

[73] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security*, 2014.

[74] E. Gökta, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Security and Privacy*, 2014.

[75] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson. ILR: where'd my gadgets go? In *Security and Privacy*, 2012.

[76] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *Security and Privacy*, 2012.

[77] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *Third USENIX Windows NT Symposium*, 1999.

[78] Intel. Intel 64 and ia-32 architectures software developers manual.

[79] K. Johnson and M. Miller. Exploit mitigation improvement in windows 8. In *Black Hat*, 2014.

[80] Jones and Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Workshop on Automated Debugging*, 1997.

[81] D. B. JongHyup Lee, Thanassis Avgerinos. Tie: Principled reverse engineering of types in binary programs. In *NDSS*, 2011.

[82] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: low-overhead protection from code reuse attacks. In *ISCA*, 2012.

[83] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *ACSAC*, 2006.

[84] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security*, 2002.

[85] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://www.suse.de/~krahmer/no-nx.pdf.

[86] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

[87] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. PEBIL: efficient static binary instrumentation for Linux. In *ISPASS*, 2010.

[88] H. Li. Understanding and exploiting Flash ActionScript vulnerabilities. CanSecWest, 2011.

[89] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security*, 2011.

[90] 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 *PLDI*, 2005.

[91] H. Macro and I. Ripoll. Linux aslr integer overflow: Reducing stack entropy by four. http://hmarco.org/bugs/linux-ASLR-integer-overflow.html, 2015.

[92] H. Macro and I. Ripoll. Linux aslr mmap weakness: Reducing entropy by half. http://hmarco.org/bugs/linux-ASLR-reducing-mmap-by-half.html, 2015.

[93] H. Marco-Gisbert and I. Ripoll. On the effectiveness of full-aslr on 64-bit linux. In *DeepSec*, 2014.

[94] A. J. Mashtizadeh, A. Bittau, D. Mazières, and D. Boneh. Cryptographically enforced control flow integrity. *CoRR*, abs/1408.1451, 2014.

[95] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security*, 2006.

[96] Microsoft. Enhanced mitigation experience toolkit (EMET). http://technet.microsoft.com/en-us/security/jj653751.

[97] V. Mohan, P. Larseny, S. Brunthalery, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, 2015.

[98] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for c. In *PLDI*, 2009.

[99] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *CGO*, 2006.

[100] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh. Foreign code detection on the windows/x86 platform. In *ACSAC*, 2006.

[101] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 2001.

[102] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[103] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[104] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *CCS*, 2013.

[105] B. Niu and G. Tan. Modular control-flow integrity. In *PLDI*, 2014.

[106] B. Niu and G. Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *CCS*, 2014.

[107] J. Oakley and S. Bratus. Exploiting the hard-working DWARF: trojan and exploit techniques with no native executable code. In *USENIX WOOT*, 2011.

[108] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *ACSAC*, 2010.

[109] V. Pappas, M. Polychronakis, and A. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy*, 2012.

[110] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security*, 2013.

[111] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. In *VEE*, 2011.

[112] M. Payer, T. Hartmann, and T. R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *Security and Privacy*, 2012.

[113] A. Prakash, H. Yin, and Z. Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *ASIACCS*, 2013.

[114] A. Prakashm, X. Hu, and H. Ying. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *NDSS*, 2015.

[115] M. Prasad and T.-c. Chiueh. A binary rewriting defense against stack based overflow attacks. In *USENIX ATC*, 2003.

[116] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. LIFT: a low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, 2006.

[117] I. Rapid7. The metasploit framework. http://www.metasploit.com/.

[118] B. S., J. Oakley, A. Ramaswamy, S. Smith, and M. Locasto. Katana: Towards patching as a runtime part of the compiler-linker-loader toolchain. *International Journal of Secure Software Engineering*, 2010.

[119] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*, 2008.

[120] F. Schuster, T. Tendyck, L. Christopher, L. Davi, A.-R. Sadeghiy, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Security and Privacy*, 2015.

[121] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A link-time optimizer for the intel ia-32 architecture. In *In Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[122] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO*, 2003.

[123] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security*, 2010.

[124] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *CCS*, 2014.

[125] F. J. Serna. CVE-2012-0769, the case of the perfect info leak, Feb. 2012. http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[126] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, 2007.

[127] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS*, 2004.

[128] R. Shapiro, S. Bratus, and S. W. Smith. "weird machines" in elf: A spotlight on the underappreciated metadata. In *USENIX WOOT*, 2013.

[129] U. I. P. L. SIG. Dwarf debugging information format. http://www.dwarfstd.org/doc/dwarf-2.0.0.pdf, 1993.

[130] 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 *Security and Privacy*, 2013.

[131] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *NDSS*, 2015.

[132] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *NDSS*, 2015.

[133] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: a new approach to computer security via binary analysis. In *ICISS*, 2008.

[134] A. Stewart. DLL side-loading: A thorn in the side of the anti-virus industry. http://www.fireeye.com/resources/pdfs/fireeye-dll-sideloading.pdf, 2014.

[135] P. team. Restricted mprotect, 2000. https://en.wikipedia.org/wiki/PaX#Restricted_mprotect.28.29.

[136] 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 *USENIX Security*, 2014.

[137] TIS. Executable and linking format (ELF) specification. http://www.uclibc.org/docs/elf.pdf, 1995.

[138] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[139] Z. Wang and X. Jiang. HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy*, 2010.

[140] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *CCS*, 2012.

[141] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *ACSAC*, 2012.

[142] wikipedia. Open addressing hashing. http://en.wikipedia.org/wiki/Open_addressing, 2012.

[143] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: runtime intrusion prevention evaluator. In *ACSAC*, 2011.

[144] xorl eax, eax. Linux kernel aslr implementation, 2011. https://xorl.wordpress.com/2011/01/16/linux-kernel-aslr-implementation/.

[145] L. Xun. LEEL: Linux executable editing library. Master's thesis, University of Singapore, 1999.

[146] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy*, 2009.

[147] H. Yin and D. Song. Temu: Binary code analysis viawhole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, 2010.

[148] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS*, 2011.

[149] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *the 18th ACM conference on Computer and communications security (CCS)*, 2011.

[150] J. Zeng, Y. Fu, and Z. Lin. Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework. In *VEE*, 2015.

[151] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, and L. Szekeres. Protecting function pointers in binary. In *ASIACCS*, 2013.

[152] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *Security and Privacy*, 2013.

[153] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, 2013.