Control Flow Integrity for COTS Binaries

Mingwei Zhang and R. Sekar, *Stony Brook University*

This paper is included in the Proceedings of the
22nd USENIX Security Symposium.

August 14–16, 2013 • Washington, D.C., USA

ISBN 978-1-931971-03-4

Open access to the Proceedings of the
22nd USENIX Security Symposium
is sponsored by USENIX

Control Flow Integrity for COTS Binaries *

Mingwei Zhang and R. Sekar
Stony Brook University
Stony Brook, NY, USA.

Abstract

Control-Flow Integrity (CFI) has been recognized as an important low-level security property. Its enforcement can defeat most injected and existing code attacks, including those based on Return-Oriented Programming (ROP). Previous implementations of CFI have required compiler support or the presence of relocation or debug information in the binary. In contrast, we present a technique for applying CFI to stripped binaries on x86/Linux. Ours is the first work to apply CFI to complex shared libraries such as `glibc`. Through experimental evaluation, we demonstrate that our CFI implementation is effective against control-flow hijack attacks, and eliminates the vast majority of ROP gadgets. To achieve this result, we have developed robust techniques for disassembly, static analysis, and transformation of large binaries. Our techniques have been tested on over 300MB of binaries (executables and shared libraries).

1 Introduction

Since its introduction by Abadi et. al. [1, 2], Control-Flow Integrity (CFI) has been recognized as an important low-level security property. Unlike address-space randomization [24, 5] and stack cookies [12, 17], CFI's control-flow hijack defense is not vulnerable to the recent spate of information leakage and guessing attacks [40, 37, 16]. Unlike code injection defenses such as DEP (data execution prevention), CFI can protect from existing code attacks such as return-oriented programming (ROP) [38, 9, 49] and jump-oriented programming (JOP) [10, 7]. In addition to exploit defense, CFI provides a principled basis for building other security mechanisms that are robust against low-level code attacks, as evidenced by its application in software fault isolation [27, 47] and sandboxing of untrusted code [15, 46].

An important feature of CFI is that it can be meaningfully enforced on binaries. Indeed, some applications of CFI, such as sandboxing untrusted code, explicitly target binaries. Most existing CFI implementations, including those in Native Client [46], Pittsfield [27], Control-flow locking [6] and many other works [22, 3, 42, 4, 36] are implemented within compiler tool chains. They rely on information that is available in assembly code or higher levels, but unavailable in COTS binaries. The CFI implementation of Abadi et al [2] relies on relocation information. Although this information is included in Windows libraries that support ASLR, UNIX systems (and specifically, Linux systems) rely on position-independent code for randomization, and hence do not include relocation information in COTS binaries. We therefore develop a new approach for enforcing CFI on COTS binaries without relocation or other high-level information.

Despite operating with less information, the security and performance provided by our approach are comparable to that of the existing CFI implementations. Moreover, our implementation is robust enough to handle complex executables as well as shared libraries. We begin by summarizing our approach and results.

1.1 CFI for COTS Binaries

We present the first practical approach for CFI enforcement that scales to large binaries as well as shared libraries without requiring symbol, debug, or relocation information. We have developed techniques that cope with the challenges presented by static analysis and transformation of large programs, including those of Firefox, Adobe Acrobat 9, GIMP-2.6 and glibc. In our experiments, we have transformed and tested over 300MB of binaries. Some of the key features of our design are:

- *Modularity:* Each shared library and executable is instrumented independently to enforce CFI. Our technique ensures that when an executable is loaded and run, CFI property is enforced globally across the executable and all the shared libraries used by it.

*This work was supported in part by AFOSR grant FA9550-09-1-0539, NSF grant CNS-0831298, and ONR grant N000140710928.

- *Transparency:* If our instrumentation made even the smallest changes to (stack, heap or static) memory used by a program, it can cause complex programs to fail or function differently. As an example, consider saved return addresses on the program stack. Since code rewriting causes instruction locations to change, a straight-forward implementation would change these saved return addresses. Unfortunately, programs use this information in several ways:
  - Position-independent code (PIC) computes the locations of static variables from return address.
  - C++ exception handler uses return addresses to identify the function (or more specifically, the try-block within the function) to which the exception needs to be dispatched.
  - 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, our instrumentation has been designed to provide full transparency.

  The principal challenge in achieving full transparency is one of performance. To address this challenge, we have developed new optimization techniques.
- *Compiler independence and support for hand-coded assembly:* Our approach does not make strong assumptions regarding the compiler used to generate a binary, such as the the conventions for generating jump tables. Indeed, our code has been tested with hand-written assembly, such as that found in low-level libraries (e.g., glibc). It has been tested with the two popular compilers on Linux, GCC and LLVM.

1.2 Quality of Protection

An ideal CFI implementation will restrict program execution to exactly the set of program paths that can be taken. In practice, due to the fact that targets of indirect control-flow (ICF) transfers are difficult to predict, CFI implementations enforce a conservative approximation of ideal CFI. Different techniques enforce different approximations, so a natural and important question concerns the relative strengths of these techniques. To answer this question, we propose a simple metric, called average indirect target reduction (AIR) which quantifies the fraction of possible indirect targets eliminated by a CFI technique. To compute AIR, we start with the fraction of possible targets eliminated by a CFI technique for each ICF transfer instruction, and average this number across all ICF transfer instructions. (See Definition 1 on Page 6.)

AIRs of several types of CFI are shown in Figure 1. For the base case of an unprotected program, every byte address in the code is a possible ICF target, and the AIR is 0%. We then define a coarse form of CFI called instr-CFI that limits ICF transfers to instruction boundaries. It eliminates attacks that jump to the middle of instructions. Bundle-CFI is another coarse form of CFI used in PittsField [27] and Native Client [46]. It limits ICF transfers to addresses that are multiples of 16 (PittsField) or 32 (Native Client).

CFI type	Description	AIR (%)
null	no CFI protection	0.00
instr	Restrict ICFs to valid instruction boundaries	79.27
bundle	Instructions grouped into 32-byte bundles [46]. All ICFs must target the start of a bundle.	96.04
reloc	CFI based on relocation information. Indirect calls/jumps to target any location present in relocation table, returns to target a location immediately following a call.	99.13
strict	Enforces property closely matching reloc-CFI but does not require relocation info.	99.08
bin	Generalizes strict-CFI to avoid special treatment of threads and exceptions	98.86

Figure 1: CFI flavors and strengths on SPEC CPU2006.

The next version, reloc-CFI, captures the strength of CFI implementation described by Abadi et al [2]. It relies on relocation information in binaries. (See Section 4.2 for more discussion).

Large and complex binaries contain several exceptions to the simple model of calls, returns and indirect jumps embodied in many CFI works:

- *Returns used as jumps.* Return instructions are some times used to jump to functions by pushing their address on the stack and returning. Examples include code for thread context switching, signal handling, etc.
- *Returns to caller function, but not a return address.* Some times, returns go back to a caller, but don't target a return address, e.g., due to C++ exceptions.
- *Jumps to return addresses.* Functions such as `longjmp` use an indirect jump that targets a return address.
- *Runtime generation of new ICF targets.* Some applications create ICF targets on the fly using `dlopen` to add additional libraries at any point during runtime.
- *Indirect jumps using arithmetic operations.* Low-level assembly code can contain ICF targets that are computed using multiple arithmetic operations.

To cope with these exceptions, our approach, called bin-CFI, avoids making any of the common assumptions regarding ICF targets in general. Instead, it relies on static analysis and a very conservative set of assumptions so that it can scale to large executables and libraries.

Note that bin-CFI eliminates about 99% of possible indirect targets. Moreover, it experiences only a small

decrease in AIR as compared to reloc-CFI. This provides evidence that *our approach achieves compatibility with COTS binaries without incurring a major reduction in its quality of protection.*

To further pinpoint the sources of the slight decrease in AIR, we implemented a stricter version of bin-CFI called strict-CFI. It uses the same binary analysis techniques as bin-CFI, but instead of providing a general way to handle exceptions and threads, it simply uses a relaxed policy for a few specific instructions in system libraries that perform thread switching or exception unwinding. Note that the strict-CFI has an AIR very close to that of reloc-CFI, pointing out that the sources of AIR decrease are the exceptions that need to be made in order to support large and complex binaries. Effective precision loss incurred by our static analysis is very small (0.05%) as compared to the use of relocation information.

1.2.1 Experimental Evaluation

We present a detailed experimental evaluation of our technique. Key points include:

- *Good performance:* Techniques for achieving transparency and modularity can exact a price in terms of performance. We describe several optimization techniques in Section 6 that have reduced the overhead to about 8.54% across the SPEC CPU benchmark suite.
- *ROP and JOP defense:* As our AIR measurements indicate, about 99% of possible ICF targets have been eliminated by bin-CFI. Moreover, on the SPEC CPU 2006 benchmark, our technique also eliminated about 93% of ROP gadgets that were found by the popular ROP gadget discovery tool ROPGadget [35].
- *Control-flow hijack detection.* Our results show that bin-CFI defeats the vast majority of control-flow hijack attacks from the RIPE benchmark [45].

2 Disassembly

2.1 Background

There are two basic techniques for disassembly: *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 an 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 even 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 ICF transfers.

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

2.2 Our Disassembly Technique

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 possible ICF targets: some of these locations don't represent code addresses. Thus, blindly following ICF targets computed by static analysis can lead to incorrect disassembly. We therefore use 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.

Our approach 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,

[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.

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, 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 our confidence regarding which of the two has been incorrectly disassembled. In the absence of additional information, our approach considers both possibilities.

Since errors in linear disassembly arise due to gaps, our error correction step relies on identifying and marking these gaps. An incorrectly disassembled instruction signifies the presence of a gap, and we need to find its beginning and end. To find the beginning of the gap, we simply walk 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, we rely on static analysis results (Section 3). 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, we extend 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 we are done. Otherwise, the whole process needs to be repeated. While it may seem that repetition of disassembly is an unnecessarily inefficient measure, we have used it because of its simplicity, and because disassembly errors have been so rare in our implementation that no repetition was needed for the vast majority of our benchmarks.

3 Indirect Control Flow Analysis

In this section, we describe a static analysis for discovering possible ICF targets. We classify ICF targets into several categories, and devise distinct analyses to compute them:

- *Code pointer constants (CK)* consists of code addresses that are computed 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 to handle exceptions.
- *Exported symbol addresses (ES)* include export function addresses.
- *Return addresses (RA)* include the code addresses next of a call.

Our static analysis results are filtered to retain only those addresses that represent valid instruction boundaries in disassembled code.

3.1 Identifying Code Pointer Constants (CK)

In general, there is no way to distinguish a code pointer from other types of constants in code. So, we take 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.
- 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 we check 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 our 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, we use a 4-byte sliding window over the code and data to identify all potential code pointer constants.

3.2 Identifying Computed Code Pointers (CC)

Whereas our 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 any degree of 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[2]. 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, we support computed code pointers in a limited set of contexts where they seem to arise in practice. Indeed, the only context in which we have observed code pointer arithmetic is that of jump tables[3].

The most common case of jump tables arise 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, this approach isn't feasible in our case since we wish to handle many low-level libraries that contain hand-written assembly code. So, we begin by identifying properties that we believe are generic to jump tables:

- Jump table targets are intra-procedural: the ICF transfer instruction and ICF target are in the same function. (We don't require function boundaries — we estimate them 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 our implementation.

Based on these characteristics, we have developed 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 we fall back to the following approach that uses information about exported function symbols. We treat the region between two successive exported function symbols as an approximation of a function. (Note that this approximation is conservative, as there may be non-exported functions in between.) We then construct a control-flow graph for each region.

In the second step, we identify instructions that perform an indirect jump. We perform a backward walk from these instructions using the CFG. All backward paths are followed, and for each path, we trace 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, we have not implemented such an analysis, especially because the index value may come from other functions. Instead, we make an assumption that valid *Ind* values will start around 0.

In the third step, we enumerate possible values for the index variable, compute the jump target for each value, and check if it falls within the current region. Specifically, we check 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.

We start from *Ind* value of 1, and explore values on either side until we reach values for which the computed target is invalid.

We point out 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, we extend 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 can fail in some 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, we can conservatively add every instruction address within the region to CC.

3.3 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 [26] format. We use an existing tool, katana [29, 30], to parse these DWARF sections and get both base addresses and offsets, and thus compute the set EH. (We point out 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.)

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

[3] C++ exception handling also involved address arithmetic on return addresses, but we can rely on exception handler information that must be included in binaries rather than the CC analysis.

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.

4 Defining and Assessing CFI for Binaries

4.1 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, our CFI enforcement 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 quality of protection offered by a CFI technique.

Definition 1 (Average Indirect target Reduction (AIR)) *Let $i_1, \ldots, i_n$ 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. 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.

4.2 A Simple CFI Property based on Relocation

CFI techniques 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[4] can replace an indirect call (IC) with 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 don't 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 IJ, 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 [2] indicates their use of relocation information, and confirms the above policy regarding ICs. No specifics are provided regarding IJs and returns, but for reasons described above, we believe that they support the reloc-CFI policy described above. We also note that indexed hooks [22] uses a single table for ICs and IJs, and another for returns, enforcing reloc-CFI but in a kernel environment.

4.3 Strict-CFI: A CFI Property for Binaries Closely Matching Reloc-CFI

Strict-CFI is derived from reloc-CFI, except that it uses ICF targets computed by our ICF target analysis 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:

- 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.

Since they apply to a very small fraction of ICF transfers in a program, their overall effect on AIR is negligible. Thus, the difference in AIR between reloc-CFI and strict-CFI will pinpoint the precision loss due to the use of static analysis in place of relocation information.

4.4 Bin-CFI: CFI for Complex Binaries

Complex binaries can contain exceptions to the simple model of ICF transfers outlined earlier. To define a suitable CFI property for such binaries, 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[5].

We are now ready to define bin-CFI as shown in Figure 2.

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

[5] 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)	Y	
Exception handling addresses (EH)	Y	
Exported symbol addresses (ES)		Y
Code pointer constants (CK)	Y	Y
Computed code addresses (CC)	Y	Y

Figure 2: Bin-CFI Property Definition

It is easy to see that strict-CFI is stricter than bin-CFI. 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. We therefore permit returns 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, 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 we have done on the second column of Figure 2.

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, we chose to merge IC and PLT to use the same table of valid targets.

5 Implementation

Although our design is largely applicable to most architectures, our implementation targets 32-bit x86 processors running Linux. For this reason, some implementation aspects discussed below are specific to this platform.

5.1 Disassembly

Binaries on Linux (and most other UNIX systems) use the ELF (Executable and Linkable Format) [25] format. We support 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 our approach utilizes the data in read-only data sections, it is mainly concerned with the code sections.

Our implementation utilizes `objdump` to perform linear disassembly. We have built our disassembly error detection and correction components on top of `objdump`. In our experience, disassembly errors occurred primarily due to insertion of null padding generated by legacy code or linker script. In addition, we discovered 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. We eliminated this error by treating these prefixes as independent instructions, so that jumps could target the instruction with or without the prefix.

5.2 Instrumentation and Regeneration of Binary

After disassembly, the resulting code is instrumented to enforce CFI. The specifics of this instrumentation are described in Section 5.3. Below we describe the generation of a binary from instrumented code, since a general understanding of this process will enable a fuller understanding of the instrumentation steps.

Instrumentation is performed on assembly representation. This simplifies our 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 of `objdump`, we associate 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 we are inserting additional code during instrumentation.)

After rewriting, the instrumented assembly file is processed using the system assembler (in our case, the GNU assembler *gas*) to produce an object file. We extract the code from this object file and then use the *objcopy* tool to inject it into the original ELF file. Note that the original code sections are not overwritten. This ensures that any attempt by the instrumented program to read its own code will produce the same results as the original program.

The final step prepares 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. The original code segments are made unexecutable. For shared libraries, it is also necessary to update the dynamic symbol sections.

5.3 Instrumentation for CFI

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 we 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[6].

The typical way to deal with this uncertainty, employed in dynamic binary translation (DBT) [8], 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

⟨original address, new address⟩

At runtime, `addr_trans`, a piece of trampoline code, performs address translation. (In fact, there are two such trampolines, one corresponding to each column of Figure 2.) Instrumentation is inserted at the site of the original indirect control-flow transfer instruction as shown in Figure 3.

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

Figure 3: 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.[7] 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. As noted earlier, there are two address translation routines, one corresponding to each column of Figure 2. The valid ICF targets for each table consists of the subset of ICF targets computed by the static analysis described in Section 3 that appear in the corresponding column of Figure 2.

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 described later in this section. After the range check, `addr_trans` performs address translation. Our implementation relies on closed hashing [44] to perform an efficient lookup of the table described above. Rather than storing just the target address `L_`A in the table, our implementation 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, translation table only contains the offsets rather than absolute addresses. Consequently, the base address of the module needs to be subtracted from the runtime address given to the translation routine. We rely 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 we need 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 x86, there are only two instructions that use PC this way: call and return. A `call X` is translated into a `push next; jmp X`, 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[8], thus ensuring that

[6] Here again, relocation information can address this uncertainty, but in our case, this is unavailable.

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

[8] In x86-64 architecture, any PC-relative data addressing needs to

program behavior is unaffected by our instrumentation.

Modularity. Support for shared libraries is achieved as follows. Our technique 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 we transform 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. We use 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. 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. We modified the source code of `ld.so` to accomplish this. Our 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. Our changes relate to about 300 lines of the source code of `ld.so`.

Our loader modification also addressed two other idiosyncrasies of `ld.so`. First, note that our 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>`. We changed the loader so that it compensates for the change in the entry point, and hence works correctly in all cases.

The second idiosyncrasy concerns the use of return instructions for lazy symbol resolving. Lazy symbol resolving is implemented 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 our CFI implementation. Instead, we chose to modify the loader to use indirect jumps instead of returns, and restricted the target of these jumps with the policy shown in Figure 2 for PLT entries.

be translated too. This can be done easily by modifying the offset value.

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

Figure 4: Optimized instrumentation of calls

Signals. Signal is another mechanism to redirect program control flow. If a program registers its signal handlers, once again we will have the problem that the program will specify the location of the handler in original code, whereas we want the signal to be delivered to the instrumented code. (This problem arises because signals are delivered by the kernel, which is not aware of the address translations used to correctly handle code pointers.)

Our 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.

6 Optimizations

6.1 Improving Branch Prediction (BP)

Modern processors use very deep pipelines, so branch prediction misses can greatly decrease performance. Unfortunately, our translation of returns (into a combination of pop and jmp) leads to misses. When a return instruction is used, the processor is able to predict the target by maintaining a stack that keeps track of calls. When it is replaced by an indirect jump, especially one that is always made from a single trampoline routine, prediction fails.

To address this problem, we modified the transformation of calls and returns as shown in Figures 4 and 5. 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. Similarly, 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.

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

Figure 5: Optimized instrumentation of returns

The key point about this transformation is that the processor sees a return in Figure 5 that returns from the call it executed (Figure 4, label `L_060b1`). Although the address on the program stack was adjusted (Figure 4, label `S_060b1`), this is reversed by address translation in Figure 5. As a result, the processor's predicted return matches the actual return address on the stack.

6.2 Avoiding Address Translation (AT)

We explored three optimizations aimed at eliminating address translation 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, we create 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 we don't know the size of the original table. Note, however, that we have a good guess, based on the CC computation technique from Section 3.2. We first check that the index variable is within this range, and if so, use the new table. Otherwise, we use 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, we 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 our implementation, we choose to apply it only to return instruction. We used profiling to determine if the return frequently targets the same location.

6.3 Violating Transparency (VT)

Using static analysis results, we can safely avoid some of the overheads associated with full transparency. The following are two optimizations we use:

VT.1 no saving of eflags

VT.2 use non-transparent calls

To achieve, VT.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 we can safely use VT.1. In fact, we discover that eflags is live only in a few jump tables.

When VT.2 is enabled, all return addresses are within the new code. Note that VT.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 non-transparent mode will not lead to any problems, as long as the offset added to obtain data address is appropriately adjusted.

7 Evaluation

We first evaluate functionality of our system, focusing on disassembly, and compatibility with different compilers. Next, we evaluate its effectiveness in terms of the AIR metric and attack defense. Then, we evaluate its runtime and memory overheads, Finally, we summarize the limitations of the approach and its current implementation.

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 6: Disassembly Correctness

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 7: Real World Program Functionality Test

7.1 Functionality

Testing transformed code. We tested the SPEC CPU2006 programs (Figure 8). This benchmark comes with scripts to verify outputs, thus simplifying functionality testing.

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`.

Altogether, we had to transform 786 shared libraries during testing. The total code transformed was over 300 MB, of which the libraries were about 240MB and executables were about 60MB. We tested each of these programs and ensured that they worked correctly. A subset of these tests is shown in Figure 7.

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

Note that multiple object files are combined by the linker to produce an executable or library. We intercept the linker `ld` to record address ranges in the code that correspond to each object file. This information is used to compare compiler-produced assembly for each object file with the corresponding part of the disassembler output.

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

Testing Code Generated by Alternative Compilers. We applied our instrumentation to two programs compiled using LLVM. In particular, we used Clang 2.9 to compile two programs in the `OpenSSH` project, `ssh` and `scp`. Experiments shows that both LLVM generated `ssh` and `scp` function correctly when we used them to login to a remote server and copy a large file to/from the server.

7.2 CFI Effectiveness Evaluation

Figure 8 compares the AIR metric for bin-CFI with strict-CFI, reloc-CFI, bundle-CFI and instr-CFI. To calculate AIR of reloc-CFI, we recompiled SPEC2006 programs using "-g" and a linker option "-Wl,-emit-relocs" to retain all the relocations in executables. We can now calculate AIR from the description of reloc-CFI in Section 4.2 and Definition 1.

To calculate AIR for bundle-CFI, we recompiled SPEC2006 using the 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 AIR number is smaller because the base is the original program size; programs compiled using Native Client tool-chain are larger 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 8: AIR metrics for SPEC CPU 2006.

7.3 Security Evaluation

7.3.1 Control-Flow Hijack Attacks

To evaluate control flow hijack defense, we used the RIPE [45] test 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. Ex-

	DEP disabled	DEP enabled
Original	520	140
CFI	90	90

Figure 9: Security Evaluation using RIPE

ploit 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 we can experiment with is that of enabling or disabling DEP.

Originally, on Ubuntu 11.10 platform, 520 attacks survive with data execution prevention (DEP) disabled. With DEP enabled, 140 attacks survive. All of these attacks are return-to-libc attacks.

The 2nd row in Figure 9 shows bin-CFI could defeat 430 attacks including 380 code injection attacks and 50 return-to-libc attacks, even when DEP is disabled. In both scenarios, when DEP is enabled or disabled, however there are 90 function pointer overwrite attacks that survive in CFI.

Code injection attacks are defeated by CFI because global data, stack and heap are not allowed targets of ICF transfers. 50 out of 140 return-to-libc attacks are defeated because they overflow return addresses and try to redirect control flow to the libc functions and violate the policy of bin-CFI. Those attacks are defeated.

The function pointer overwrite attacks that succeed are 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 our 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 any function whose address is taken.

7.3.2 ROP Attacks

We use the tool ROPGadget-v3.3[35], an ROP gadget generator/compiler, as our testing tool. It scans binaries to find useful gadgets for ROP attacks.

Figure 10 shows that CFI enforcement is effective, resulting in the elimination of the vast majority (93%) of gadgets in the original program. Moreover, there is little diversity in the gadgets found — the tool was able to find only the following gadgets:

- `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%)

There is little variety in these gadgets. Among other missing features, note the complete lack of useful arithmetic operations in the identified gadgets. As a result, the tool was unable to build even a single exploit using these 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 10: Gadget elimination in different CFI implementation

7.4 Performance Evaluation

Our testbed consists of an Intel core-i5 2410m CPU with 4GB memory, running Ubuntu 11.10 (32-bit version), with glibc version 2.13. We used the SPEC 2006 CPU benchmark to evaluate both the runtime overhead and space overhead.

7.4.1 Runtime Overhead

Figure 11 shows the runtime overheads of CFI enforcement on SPEC CPU 2006 benchmarks. The average overhead for C programs is 4.29%. Due to C++ exception handling, VT.2 (Section 6.3) cannot be applied to C++ programs. As a result, the overhead for C++ programs increases to an average of 8.54%. omnetpp, soplex, and povray are particular contributors to this increased overhead. One way to bring these overheads down (to match the overhead for C-programs) is to update the exception handling metadata to use code addresses within instrumented code.

7.4.2 Space and Memory Overhead

Our instrumentation introduces a new code section that is on average 1.2 times the original code size. The new

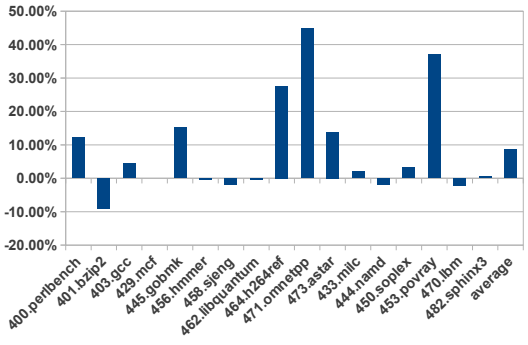

Figure 11: SPEC CPU2006 Benchmark Performance

data section introduced contains address translation table for indirect branch instructions. In total, the space overhead for bin-CFI is 139% over the original file size. Note that although the file size has increased, execution will be confined to the new code. Except in the case of programs that store read-only data in their code, other programs don't access their code even once. Hence the runtime memory overhead is unaffected by the presence of the original copy of code. Indeed, our measurements showed a very small increase in resident memory use (about 2.2% on average).

7.5 Limitations

Dynamic code. Since we rely on static transformation of binaries, any usage of dynamic code such as just-in-time compilation cannot be handled by bin-CFI. This also applies to any binary that modifies itself. These limitations are shared by most previous implementations of CFI.

Obfuscated code. Reliable static disassembly of obfuscated code is a challenging problem without satisfactory solutions. However, obfuscation is typically used on malware, whereas our target consists of benign (but possibly vulnerable) programs.

Return-into-libc attack. In general, CFI does not eliminate the threat of all return-to-libc attacks, a fact that holds true in our implementation as well.

Most return-into-libc fall into one of the two following types. The first type chains a sequence of library function calls, and relies on the semantics of these functions to perform attacks [28]. The second type relies on the side effects of library functions to realize Turing-complete ROP [41]. Both types rely heavily on returning to exported functions in glibc, and hence are defeated by bin-CFI. (Note that exported functions are excluded from allowable return targets by our policy.) However, it may be possible to construct return-to-libc attacks that make use of code pointers in glibc (or other shared libraries), or more generally, any address computed by our static analysis. These attacks could be mitigated by further tightening the policy for returns, improving the precision of static analysis, or both. We point out that even without these improvements, bin-CFI degrades return-to-libc attacks in much the same way as it degrades ROP attacks: it reduces the number of possible functions that can be used in an attack.

8 Related Work

8.1 ROP Attacks and Defenses

Return Oriented Programing (ROP) [38] is a powerful code reuse attack. It has become a very popular means to carry out successful attacks in spite of DEP. Although ROP was originally thought to be applicable primarily to CISC processors such as the x86, subsequent work has demonstrated their effectiveness on RISC architectures as well [9]. ROP attacks can target user programs as well as the kernel [19]. The introduction of JOP [10, 7] eliminates the need to use return instructions to effect ICF transfers, thereby defeating defenses that rely on the use of (repeated) returns [11, 14, 32].

Some of ROP defenses [31, 23] modify the code generation process to ensure that there are no useful gadgets in a generated binary. As they work at the level of code generation, they require source code. Rather than eliminating gadgets, some recent works [18, 43, 33] rely on fine-grained randomization that makes it difficult to find the location of useful gadgets. Instruction Location Randomization (ILR) [18] randomizes instruction locations, thereby making ROP hard. A benefit of their approach is that they can randomize return addresses, which significantly reduces the number of valid ICF targets, as return addresses constitute a majority of them. But this randomization can cause problems in large and complex binaries where a return instruction may be used for purposes other than returning from a call, e.g., PIC code data access, or to implement context-switching-like functionality.

A drawback of ILR is high space overhead. Binary Stirring (STIR) [43] solves this issue by randomizing basic blocks at load time using static rewriting. It achieves better runtime performance and reasonable space overhead. However, neither ILR nor Binary Stirring apply their work on libraries or large binaries. [33] uses static in-place randomization (IPR) to eliminate gadgets. The runtime overhead is almost zero, though the effectiveness depends on the target binary layout. In particular, a significant fraction of gadgets remain, thus limiting protection against ROP attacks.

While strong randomization could confuse attackers at runtime, and further reduce the number of usable gadgets, we have refrained from adding randomization to our technique for several reasons. First and foremost, we believe that one of principal reasons behind the success of

CFI is that it provides deterministic protection, thus laying a solid foundation for other protection mechanisms such as SFI or policy enforcement on untrusted code. Second, randomization defenses are already widely deployed in the form of ASLR and stack cookies. To the extent their randomization isn't defeated, they can provide excellent protection in conjunction with our CFI. If, on the other hand, we assume that randomization of ASLR can be defeated, then there is no good reason to believe that a randomization component added to a CFI technique won't be defeated either. Thirdly, the utility of randomization is increasingly called into question by advances in information leakage attacks. Recent exploits [37, 16] show that strong information leakage attack could help bypass ASLR with high entropy. Moreover, just-in-time code reuse attacks [39] discover gadgets using repeated information leakage attacks and are able to defeat even fine-grained code randomization.

8.2 Control Flow Integrity

Control-flow integrity (CFI) was introduced by Abadi et al [1]. 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 et al. [22] 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) [6] improves significantly on the performance of Abadi et al, 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 [13] 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, our approach emphasizes 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 our technique.

CCFIR [48], like the work presented in this paper, 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 rewriting. Moreover, since relocation information effectively identifies all code pointers, they can avoid runtime address translation, which enables them to achieve better performance. The flipside 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 [27] implements SFI on top of instruction bundling, a weaker CFI model. XFI [15] presents techniques that are based on CFI and SFI to confine untrusted code in shared-memory environments. Zeng et al [47] improve the performance of SFI using CFI and static analysis. Native client [46] is aimed at running native binaries securely in a browser context, and relies on instruction bundling. PittSFIeld, Native Client, and many other works [22, 3, 4, 42, 21, 36, 34, 20] that enforce CFI rely on compiler-provided information and even hardware support. In contrast, bin-CFI operates on COTS binaries without support from compiler, OS or hardware.

9 Conclusions

In this paper, we developed a notion of control-flow integrity that can be effectively enforced on binaries. We developed analysis techniques to compute possible ICF targets, and instrumentation techniques that limit ICF transfers to these targets. The resulting implementation defeats most common control-flow hijack attacks, and greatly reduces the number of possible gadgets for ROP attacks. We presented a robust implementation that scales to large binaries as well as complex, low-level libraries that include hand-coded assembly. Our technique is modular, supporting independent transformation of shared libraries. It also provides very good performance.

Our results realize one of central benefits of the CFI property, i.e., it can be applied to protect low-level code that is available only in the form of binaries. Although the lack of high-level information can degrade the precision of static analysis, our results demonstrate that the reduction is small; and overall, there is only a modest reduction in the strength of protection as compared to previous techniques that required source code, relocation information, or relied on compiler-based implementations.

10 Acknowledgements

We are very grateful to the developers of Katana, especially James Oakley for his quick and very helpful responses to our questions. Also we thank Edward Schwartz for his technique support.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *the 12th ACM conference on Computer and communications security (CCS)*, 2005.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, (1), Nov. 2009.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *the 29th IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[4] 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 *the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2011.

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *the 12th conference on USENIX Security Symposium*, 2003.

[6] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

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

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *the 15th ACM conference on Computer and communications security (CCS)*, 2008.

[10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *the 17th ACM conference on Computer and communications security (CCS)*, 2010.

[11] 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.

[12] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *the 7th conference on USENIX Security Symposium*, 1998.

[13] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nrnberger, and A. reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

[14] L. Davi, Ahmad-Reza Sadeghi, and M. Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[15] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[16] C. Evans. Exploiting 64-bit linux like a boss. http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html.

[17] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *the 10th conference on USENIX Security Symposium*, 2001.

[18] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *the 33th IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[19] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *the 18th conference on USENIX security symposium*, 2009.

[20] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: low-overhead protection from code reuse attacks. In *the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[21] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *the 11th conference on USENIX Security Symposium*, 2002.

[22] 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*, (4), Dec. 2011.

[23] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *the 5th European conference on Computer systems (EuroSys)*, 2010.

[24] the PaX team. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt, 2001.

[25] Tool Interface Standard. Executable and linking format (ELF) specification. http://www.uclibc.org/docs/elf.pdf, 1995.

[26] UNIX International Programming Languages SIG. DWARF debugging information format. http://www.dwarfstd.org/doc/dwarf-2.0.0.pdf, 1993.

[27] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *the 15th conference on USENIX Security Symposium*, 2006.

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

[29] J. Oakley and S. Bratus. Exploiting the hard-working DWARF: trojan and exploit techniques with no native executable code. Technical report, Computer Science Department, Dartmouth College, 2011.

[30] J. Oakley and S. Bratus. Exploiting the hard-working DWARF: trojan and exploit techniques with no native executable code. In *the 5th USENIX conference on Offensive technologies (WOOT)*, 2011.

[31] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[32] V. Pappas. kBouncer: Efficient and transparent ROP mitigation. Technical report, Columbia University, 2012.

[33] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *the 33th IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[34] A. Prakash, H. Yin, and Z. Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *the 8th ACM SIGSAC symposium on Information, computer and communications security (ASIACCS)*, 2013.

[35] J. Salwan. ROPGadget. http://shell-storm.org/project/ROPgadget.

[36] 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 *the 19th conference on USENIX Security Symposium*, 2010.

[37] F. J. Serna. CVE-2012-0769, the case of the perfect info leak, 2012.

[38] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *the 14th ACM conference on Computer and communications security (CCS)*, 2007.

[39] 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 *the 34th IEEE Symposium on Security and Privacy*, 2013.

[40] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *the 2nd European Workshop on System Security (EUROSEC)*, 2009.

[41] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *the 14th international conference on Recent Advances in Intrusion Detection (RAID)*, 2011.

[42] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *the 31th IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[43] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *the 19th ACM conference on Computer and communications security (CCS)*, 2012.

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

[45] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: runtime intrusion prevention evaluator. In *the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[46] 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 *the 30th IEEE Symposium on Security and Privacy (Oakland)*, 2009.

[47] 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.

[48] 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 *the 34th IEEE Symposium on Security and Privacy*, 2013.

[49] D. D. Zovi. Practical return-oriented programming. Technical report, SOURCE, 2010.