Control Flow and Code Integrity for COTS binaries:

An Effective Defense Against Real-World ROP Attacks

Mingwei Zhang *
Privacy and Intelligence Lab
Intel Labs
mingwei.zhang@intel.com

R. Sekar
Stony Brook University
Stony Brook
sekar@cs.stonybrook.edu

ABSTRACT

Despite decades of sustained effort, memory corruption attacks continue to be one of the most serious security threats faced today. They are highly sought after by attackers, as they provide ultimate control — the ability to execute arbitrary low-level code. Attackers have shown time and again their ability to overcome widely deployed countermeasures such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) by crafting Return Oriented Programming (ROP) attacks. Although Turing-complete ROP attacks have been demonstrated in research papers, real-world ROP payloads have had a more limited objective: that of disabling DEP so that injected native code attacks can be carried out. In this paper, we provide a systematic defense, called Control Flow and Code Integrity (CFCI), that makes injected native code attacks impossible. CFCI achieves this without sacrificing compatibility with existing software, the need to replace system programs such as the dynamic loader, and without significant performance penalty. We will release CFCI as open-source software by the time of this conference.

1. INTRODUCTION

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 the most widely deployed security feature despite its well-known weakness against code reuse attacks such as return-to-libc [42]. Subsequent to its deployment, attackers have become increasingly skilled at attacks that reuse existing code in order to bypass DEP. Introduction of Return Oriented Programming (ROP) [51] vastly expanded the scope of code-reuse attacks. By crafting small snippets of existing code (called "gadgets"), attackers could perform arbitrary (i.e., Turing-complete) computation. While Turing completeness is an attractive property that is often the topic of research papers, real-world ROP attacks have had a more limited objective: stringing together a small set of gadgets to bypass DEP.

Control flow integrity (CFI) [16] is a well-known countermeasure against control-flow hijack attacks. Such attacks typically rely on memory corruption vulnerabilities to overwrite code pointers, e.g., return addresses on the stack, or variables containing function pointers. CFI thwarts most such attacks by constraining the targets of indirect control-flow transfers to be consistent with a statically computed control-flow graph. Many research efforts have demonstrated that CFI can be implemented within a compiler [46, 44] or on binaries [66, 63], and moreover, can be applied to large and complex software packages such as web browsers [57].

ROP attacks, as originally proposed, require repeated subversion of control-flows, and moreover, use indirect branches that target the midst of instructions. As a result, they can be stopped by CFI. However, recent research efforts have shown that stealthy ROP attacks can be devised that use only those targets that are permitted by CFI [25, 32]. These attacks rely on the limited precision of practical CFI techniques, e.g., the ability of returns to target any instruction that follows a `call` instruction. In addition, researchers have developed many evasive ROP attacks [31, 22] that thwart other types of ROP defenses as well.

A natural approach for defeating these stealthy ROP attacks is finer-granularity CFI. Unfortunately, increased precision typically leads to false positives, especially on complex applications — false positive mitigation was the reason why coarse-grained CFI approaches were proposed in the first place. Moreover, precise static analyses required for finer granularity CFI are difficult to achieve on binaries, thus precluding its application to low-level code written in assembly and third-party libraries. Finally, fine granularity CFI won't stop all ROP because of precision loss inherent in CFI approaches, either due to static analysis [16], or due to the need to support dynamic loading [66, 63, 46].

The above discussion leads to the conclusion that while measures can be developed to make ROP harder, they are unlikely to be eliminated as a means for code-reuse attacks. We therefore propose a hybrid approach that not only targets the means, but also the effects targeted by real-world ROP. The one effect targeted by all real-world ROP attacks that we are aware of, as well as the stealthy attacks proposed against CFI techniques, is that of executing injected native code. In other words, these are all code-reuse-to-code-injection (CRCI) attacks. This is because real-world ROPs have to overcome a range of difficulties:

- *Limited variety of gadgets.* Due to defenses such as ALSR, frequent software updates, customized compiler optimizations, version changes and others, some gadgets are elim-

*This work was completed when he was at Stony Brook University
†This work was supported in part by grants from NSF (CNS-0831298 and CNS-1319137) and ONR (N00014-15-1-2378).

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA

ISBN 978-1-4503-3682-6/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818016

inated, 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 maximum size of ROP payload is limited by the specifics of exploit context. An analysis of the popular Metasploit framework demonstrates this limit: the average maximum payload across 946 vulnerabilities was only 1332 bytes [29]. This constrains the length of ROP chain.

As such, it becomes impractical to achieve the entire attacker's objective in the code-reuse phase of the attack. Instead, attackers use ROP to disable DEP[1]. Based on this observation, we develop Control Flow and Code Integrity (CFCI), an effective countermeasure to CRCI attacks.

1.1 CFCI Property and Approach

The DEP mechanism provided on today's platforms is targeted at native code injection attacks, but is secure only against a weak attacker model. In particular, it is ineffective against attackers that can carry out nontrivial code reuse attacks. We therefore develop a stronger defense that prevents escalation of attacks from code-reuse to code-injection (CRCI). Specifically, our approach enforces the following:

- *A program can load only a specified set of executable files.*
- *Only legitimate instructions in these files can ever be executed.*
- *No change can be made to any instruction during runtime.*

The first property is ensured by enforcing a code loading policy that prevents attackers from loading/launching arbitrary code. The monitor allows an executable program to have flexible policies to support program debugging, testing and running in special contexts. The second property is ensured by a non-bypassable state model that governs module loading logic. This state model is protected by the underlying CFI and other techniques, and defeats low level memory attacks that try to subvert code-loading logic. The last property is achieved using code instrumentation that handles legitimate code modifications requests without really modifying the executable code pages.

Together, these properties make native code injection impossible. We note that techniques such as write-protecting binary files (or code-signing) serve to preserve the integrity of code on the disk, but are not intended to protect code integrity in main memory. Our approach thus provides an important missing piece to ensure end-to-end code integrity.

We call our approach Control Flow and Code Integrity (CFCI), 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 (i.e., data pages containing attacker provide payload) executable.
- CRCI attacks can load malicious or vulnerable libraries.
- Loader subversion attacks can load malicious code by exploiting vulnerabilities in the dynamic loader.

A more complete list of possible attacks thwarted by CFCI appears in Section 2.

While CFCI aims to ensure code integrity, control flow integrity as an underlying feature ensures that the policies enforced by CFCI can't be bypassed. Code integrity property, in turn, protects control-flow integrity by preventing CFI checks from being bypassed by newly introduced native code.

[1]Such an ROP payload typically needs to make a system call to add execute permission to a data page.

1.2 Contributions

Our system 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, our contributions are:

- *An effective code integrity primitive for COTS binaries* that defeats real-world ROP attacks by eliminating the possibility of DEP bypass in general. It is compatible with complex COTS binaries, and does not requires changes to system software such as the loader or compilers. It works with statically-linked as well as dynamically-linked binaries. Finally, it can defeat powerful adversaries that have compromised multiple execution threads.
- *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 our design is the simplicity of this model, which increases confidence in its correct enforcement of policies for secure loading. Moreover, our approach *does not require either trusting or modifying the dynamic loader.*
- *A library loading policy* that ensures that all loaded modules have previously been transformed to enforce CFI. Moreover, library search paths and/or individual libraries can be limited for each application to prevent loading of unauthorized libraries.
- *Low performance overhead.* Our approach adds a small overhead over the base platform (PSI [64]) for application start-up, and very low overheads at runtime.
- *COTS binary friendly SFI protection.* Although there have been several x86 implementations of SFI [30, 60, 37, 62], they all required some level of compiler support. To our knowledge, the approach presented in this paper has the best compatibility for existing COTS software.

2. THREAT MODEL AND ATTACKS

Our threat model considers remote attackers that are able to interact with network-facing applications. In particular, attackers in our threat model can:

- defeat ASLR using information leakage (or other) attacks,
- hijack *more than one* thread in the victim process using code reuse attacks, and
- use these attacks to either (a) read/write/execute arbitrary memory of the victim process, subject to page protection settings, or (b) load new code, e.g., malicious or vulnerable libraries.

Below, we list the possible attacks that may be carried out by such attackers. Most of these attacks rely on data corruption and/or code-reuse attacks that remain possible despite protections such as CFI, ASLR and DEP.

2.1 Direct Attacks

In this case, the exploit code 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 attacks, many current exploits rely on this approach.

2.2 Loader Subversion Attacks

Successful CRCI 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.

2.2.1 Code-reuse attacks

These are the simplest forms of loader subversion, 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].

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 [8, 6, 7] 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 [1, 11]. This may lead to privilege escalation attacks [10, 4, 9, 5]. On Windows, FireEye reports [56] an increasing use of the WinSxS side-by-side assembly feature [15] 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 [2]. Malformed binaries are one of the best ways to trigger them [3, 12]. It has been reported that malformed relocation data can be used to circumvent code-signing on certain Apple iOS versions [13]. Researchers have also documented more general attacks, showing how malformed relocation information can be utilized to perform arbitrary operations [53].

Corrupting loader data. Some binaries require the loader to perform relocation. It is possible to exploit this capability to modify existing code. In particular, we 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 attempt to modify already loaded code. Note that some details of our attack is similar with a previous paper [53].

We 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.

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 identify three of the most 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.

2.2.4 Text Relocation Attack

Through our experiment, we found a new code injection attack using relocation metadata. In particular, we find that adding a text relocation flag can fool the loader into "repatching" the executable sections of a library. Moreover, by corrupting the loader's data structures, we could replace the original relocation table and symbol table with forged, malicious tables. When the loader performed relocation using these forged tables, it resulted in a code injection attack.

This attack is available not only on typical Linux distributions, but also other loader implementations such as those packaged with bionic libc for Android, uClibc for embedded systems and other POSIX-compliant libc such as musl-libc. It affects not only Linux, but also iOS (before v7.1) and Android (before v4.3). Further details of the attack can be found in our technical report [67]

3. SYSTEM DESIGN

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

[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.

While the design of CFCI is independent of underlying CFI, our implementation builds on the static binary instrumentation platform PSI [65], which implements BinCFI [66].

CFCI secures the loader using a small reference monitor that operates by intercepting key operations relating to code loading. This reference monitor is based on a state model of a loader described below.

3.1 Loader State Model

Our 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 CFCI 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 our 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 3.5 our design of this protected memory.

Figure 1 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.
- 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.

3.2 State Model Enforcement

CFCI maintains the current state of an ongoing load, and permits only those operations that are legal in that state. 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.

3.2.1 Checking library open operation

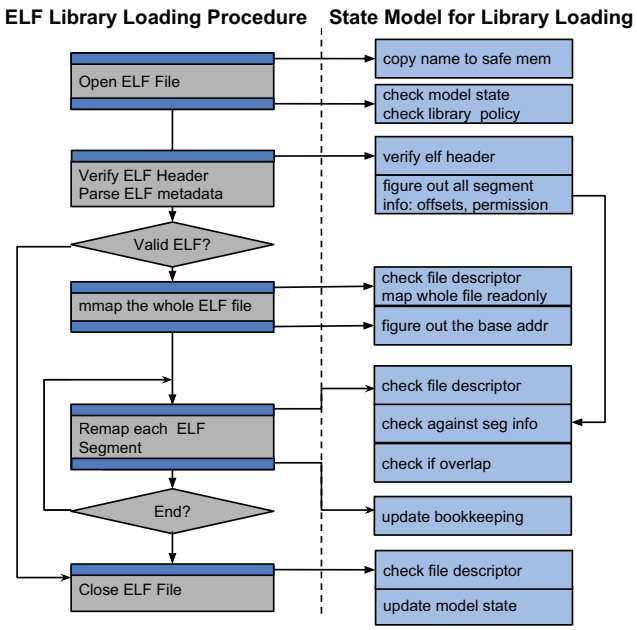

Figure 1: State Model for Module Loading

CFCI 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 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, the policy will ensure that all loaded libraries are from a predefined set. However, in practice, the exact set of libraries needed may not be known until runtime, especially for many graphical programs. To simplify policies for such applications, CFCI can permit loading of files from specified directories such as `/lib` and `/usr/lib/*`.

We configured PSI to load only libraries transformed for CFI. If a process attempts to load an untransformed library, PSI transforms it before loading.

3.2.2 Checking operations to map files 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 associations between file names and file descriptors, and is populated by the state model in Step 1. Any attack that invalidates this association can compromise the library loading policy, and hence CFCI guards against such invalidation. Ultimately, any such invalidation must happen through a call to `close`[4], so our state model intercepts this operation, and deletes the corresponding file-descriptor from its table. This prevents any subsequent use of that descriptor in `mmap` operations.

3.2.3 Checking segment boundaries

The read operation of the loader in Step 2 is intercepted and modified so that its results will be stored into protected memory. CFCI then parses this ELF metadata to obtain information about segments and where they should be loaded. A copy of this data is then returned to the loader.

In Step 4, the information saved about segment offsets

[4]Functions such as `dup` also end up calling `close`.

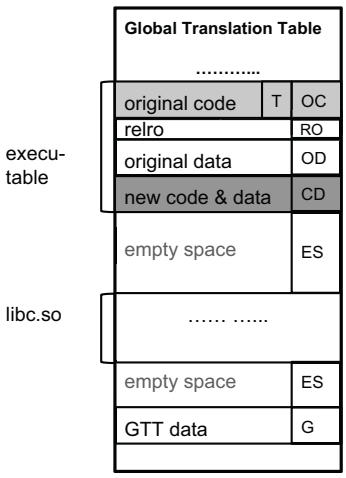

Figure 2: Layout of Memory Map Table

will be used to validate requests to map segments of the library into memory. In particular, CFCI 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 data structures holding ELF metadata, they will not be able to circumvent CFCI.

3.3 Code Integrity Enforcement

Our state model ensures that code is safely loaded from disk to memory. To ensure its continued integrity, CFCI maintains a table of relevant memory segments as shown in Figure 2, and enforces policies on operations that modify their permissions. This table consists of several segments, including the original code (non-executable), the original data, (new) instrumented code (executable) and data. Note that "`relro`" is a special data region that contains important code and data pointers. It is first made writable by the loader, then "patched" and then made read-only. The following policies are enforced on these segments:

- *No changes, or make read-only:* This policy is applied to original code (OC), new code and data (CD), memory-mapped data used by CFCI (G), and virtual DSO (VDSO — used to support fast system calls).
- *Cannot make executable:* This policy applies to original data (OD) (unless JIT is enabled — see Section 3.6), and empty space (ES).
- *One-time writable:* Original code with text relocation (OCT) and relro data (RO) can be writable (but not executable). However, once it is writable, it can only be changed back to read-only (non-executable).

3.3.1 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 our policy since only (non-executable) original code will be patched. In order to execute correctly, the transformed code needs to make use of this patched-up constant value. This is accomplished as shown in Figure 3. Suppose that the original code moves a constant to the `eax` register, and this constant is fixed up during the text relocation phase. The

Original Code:

```
mov $const, %eax
...  ...
```

Transformed Code:

```
          call _next
   _next: pop %eax
          add $offset, %eax
          mov (%eax), %eax
```

Figure 3: Patching for Text Relocation

transformed code retrieves this constant value using PC-relative addressing. Specifically, it first uses that `call`/`pop` sequence to retrieve the PC-value into `eax`. Then it adds an offset that captures the distance between the original and transformed versions of the instruction being transformed. Note that now, `eax` will point to the location of the patched-up constant value. The last step in the transformed code is to load the contents of this location into `eax`.

Using the above transformation, transformed code with text relocation could be correctly executed. Note that our transformation defeats the primary objective of text relocation, namely, avoiding the overhead of PIC code. We accept this as a reasonable trade-off for achieving compatibility, especially because text relocations not frequently used.

3.4 Library Loading Policy

Library loading policy ensures that each executable can only load a set of dependent libraries. For some high-profile applications, identifying the specific set of libraries may be well worth the effort. Tools such as `ldd` can be effective for statically enumerating the library set, especially for command-line applications. For many other applications, however, this strict policy can impact usability:

- 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 alternative libraries to be loaded.

To address these, we permit a more relaxed policy that allows libraries to be loaded from a specified set of directories.

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. Our 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.

In particular, `LD_LIBRARY_PATH` and `LD_PRELOAD` values will be checked by our specially designed library (details in Section 4) at load time, and unauthorized values that violate library loading policy will be rejected.

3.5 Protected Memory

Among SFI approaches [37, 60, 62, 30], the most efficient one for our platform (32-bit x86) relies on segmentation. Segmentation is also available for 32-bit applications on 64-bit x86 processors, and is the option currenly used by NaCl.

CFCI SFI design is similar to vx32 [30]. At the time of loading an executable, CFCI reserves a region of memory to be protected by segmentation. Specifically, we set aside the top few MBs of the lower 3GB of address space for SFI protection. This means that the base address of segments

such as `ds`, `es` and `ss` will be 0, and the limit will be something like `0xbfbfffff`. Using a base address of zero provides maximum compatibility with existing code, as it avoids any need to adjust pointers in memory, or when passing data to the kernel.

By default, the OS maps the program stack at a high address, and often, this may overlap with the region we want to set aside. To resolve this conflict, CFCI relocates the stack at process startup time, and then sets aside the high memory region for SFI protection. Next, CFCI initializes the segments. Note that segment descriptors are maintained in two tables in kernel space, LDT and GDT. Our implementation uses index 7 in GDT, 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 further modifications to these entries.

For 64-bit applications on the x86-64 architecture, we use randomization to realize protected memory. CFCI ensures that protected memory accesses could only be done through the unused TLS register (`%gs`)[5] with offset. Since segment base address is stored in kernel, memory leaks can't be used to reveal the location of protected memory, thus strengthening our randomization based defense.

3.6 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 so far already supports the first case of dynamic code. The second case, namely JIT code, poses some difficulties, and we explain 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 we consider, 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 CFCI. Naturally, code pages generated by such a JIT compiler cannot be protected from modifications, but our 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 [55] 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 an untrusted process, where the page is just readable and executable.

CFCI is compatible with this secure JIT code generator design as well. Each time new code is generated in the SDT process, it is instrumented by CFCI.

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

JIT code tends to change frequently, and the changes are typically small and localized. To obtain full benefits of JIT code, instrumentation needs to be performed in an incremental fashion. This requires some fine-tuning in the SDT to ensure that CFCI works correctly. However, since the source code of secure dynamic code generator [55] is not available, we have not pursued this incremental design yet.

4. IMPLEMENTATION

The state model and library loading policy are implemented using code instrumentation on the dynamic loader (`ld.so`). We have instrumented code logic in `ld.so` that is used for code loading. In addition, we have instrumented all the system calls that are located in `ld.so`, `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, we use the 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 will work immediately when the library is loaded.

To protect the special library from control flow and data attacks, our design 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 our library from data attacks, our checks only use protected memory mentioned in Section 3.5. Finally, our special library uses its own stack in protected memory.

5. EVALUATION

We implemented our system CFCI on top of an open source CFI tool, PSI [64]. Our test environment is Ubuntu 12.04 LTS 32bit, with Intel i5 CPU and 4GB memory.

5.1 CPU Intensive Benchmark

Since CFCI does not introduce significant additional operations at runtime, one would expect that it does not introduce significant overhead. Specifically, the only additional overhead of CFCI 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, we would expect the overhead to be small.

To validate this assessment, we evaluated CFCI with SPEC 2006 (Figure 4) using the reference dataset. The overall overhead of CFCI we observed is 14.37%, which is an addition of just 0.17% over that of our base platform PSI.

Note that PSI has a higher overhead than its initial system BinCFI. This is because PSI disables some optimizations in BinCFI such as "violating transparency" and profile-based optimization (AT.3).

5.2 Micro-benchmark for Program Loading

Runtime overheads arise chiefly from the following: (1) larger size of instrumented binaries, (2) checks performed in the context of the state model, and (3) checking of library loading policies. Almost all of these overheads occur at the time of loading a library, so we focus on load-time overhead in this experiment. We wrote a small program that loads a number of randomly chosen libraries. We measured the runtime needed to load the original version and compared

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 7: Effectiveness Evaluation of CFCI

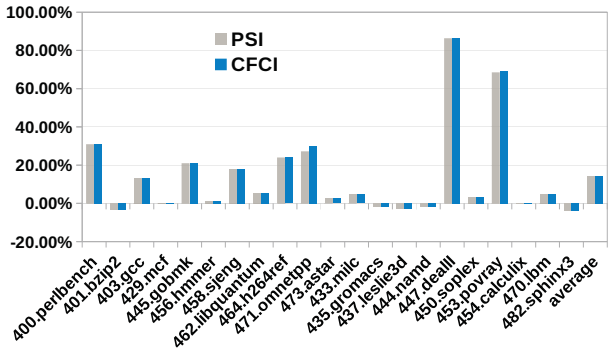

Figure 4: Performance of CFCI on SPEC2006

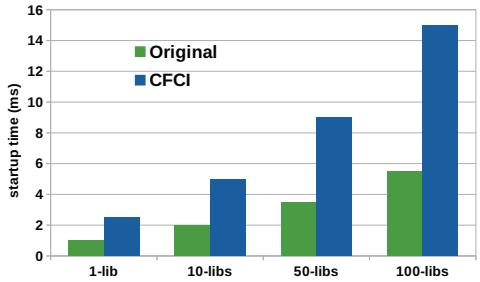

Figure 5: Micro Benchmark Evaluation of CFCI

it with that for loading the instrumented versions[6]. Figure 5 demonstrates that the overhead grows linearly with the number of libraries loaded. On average, the overhead of CFCI on a library load is 150%. While this may seem considerable, it should be kept in mind that this is a microbenchmark, and the actual overheads, when the overall execution time is considered, is much smaller.

5.3 Commonly Used Linux Applications

Since only code allocation and deallocation will generate performance overhead, we focus our evaluation on program startup when large number of modules get loaded. The results are shown in Figure 6, where the base load time is in seconds. Note that PSI has observable startup overhead (147% on average) because this phase is not optimized. The

[6]Note that we instrumented all libraries when testing transformed binary.

Program Name	Base load time	PSI overhead (over base, %)	Added CFCI 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 6: Startup Overhead on Typical Applications

additional overhead generated by CFCI is small — ranging from 2% to 18%, with an average of 8%. Note that many of these programs load more than 100 shared libraries each.

5.4 Running with Dynamic Code

To demonstrate compatibility with dynamic code, we used LibJIT [14], an actively maintained JIT engine similar to the LLVM backend. To make LibJIT compatible with CFCI, we force LibJIT to generate only non-writable code. To evaluate our performance overhead for LibJIT, we reuse an open source benchmark tool [19]. The benchmark tool is a simple program that computes the greatest common divisor (GCD). LibJIT allows dynamic functions to be invoked directly (LibJIT-fast). Our evaluation shows that CFCI overhead is 14.6%. This overhead includes about 10% overhead incurred for runtime code transformation.

5.5 Effectiveness Evaluation

Although CFI bypass technique is emerging as shown in recent research work [25, 32], real-world exploits have not been designed with CFI in mind. Therefore, running those exploits won't show the benefits of CFCI. For this reason, our effectiveness evaluation uses a combination of manual analysis based on studying the relevant CVE reports, and proof-of-concept exploits that we created ourselves. This evaluation is summarized in Figure 7.

According to our threat model in Section 2, we classify all attacks into direct attacks (`direct`), loader data corruption attacks (`ldr.data`), and loader code reuse attacks (`ldr.cr`). From Table 7, it is clear that all types of attacks are defeated. In particular, our 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, our library loading policy properly defeats

```
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 8: Library Loading Policy for Adobe Reader

all disallowed modules being loaded from unintended paths. More advanced attacks such as code reuse attacks targeting the loader are stopped by our loader state model.

5.5.1 Case study: Library policy for Adobe Reader

As described in Section 2, an attacker may attempt to load a malicious library by specifying it by name, or by corrupting the load path ("library hijacking"). CFCI 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 8 which shows our policy for Adobe Reader, a favorite target for library loading attacks [8, 6, 7].

In addition, the example also illustrates the flexibility provided in our 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 8 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.

5.5.2 Case study: Library policy for static binary

It is a common misconception that static binaries won't have any ability to load libraries. Our experiments prove otherwise. We wrote a small, statically-linked program that contains a traditional buffer overflow vulnerability. We then crafted an exploit for this vulnerability. Instead of calling `dlopen`, a function that is unavailable in statically linked binaries, our exploit redirected control to `_dl_open`, an internal function statically linked from loader. This function was passed the name of a library in `/tmp`. Our 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.

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

5.5.3 Case study: Text relocation attack

In order to evaluate code-reuse and data corruption attacks on the loader, we implemented a proof-of-concept exploit that leverages text relocation.

Our 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 is described in [67]. Our experiments shows that this exploit is very robust and works on any dynamically linked ELF programs.

We then attempted the exploit in the presence of CFCI. The attack was detected and blocked by our loader state model.

5.5.4 Case study: Making stack executable

Similar to relocation attack, we 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 sum, our exploit code bypassed these two checks and made the stack executable.

When run in the presence of CFCI, this exploit is blocked by our policies. 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 PSI alone could already defeat it. This is because other underlying CFI implementation may rely on DEP. For instance, Abadi CFI [16] requires stack to be non-executable to prevent jump-to-stack attack that has valid ID in payload.

6. RELATED WORK

Memory corruption defenses and CFI. Memory corruption attacks have been the most important targets for attackers. The most complete defense against these attacks would be based on bounds-checking [34, 59, 61, 17, 39]. Unfortunately, these techniques introduce considerable overheads, while also raising significant compatibility issues. LBC [33] achieves lower overheads while greatly improving compatibility by trading off the ability to detect non-contiguous buffer overflows. Code pointer integrity (CPI) [35] significantly reduces overheads by selectively protecting only those pointers whose corruption can lead to control-flow hijacks.

The most widely deployed defenses against memory corruption have been based on randomization [47, 21, 36, 20, 38, 18, 23, 26]. Unfortunately, randomization techniques have been repeatedly proven to be vulnerable against determined adversaries [27, 52, 54, 18]. In addition, some of the above techniques have limitations when applied to large and complex applications due to difficulties in static analysis [20, 38], or performance [26]. Nevertheless, the wide deployment of ASLR, together with DEP has raised the bar for exploit development. To bypass DEP, today's exploits employ a code reuse attack as a prelude to code injection. This code reuse attack phase has now become the new battlefield for many low level attacks and defenses.

Control flow integrity [16] provides a foundation for defending against code-reuse attacks. Several recent research efforts [66, 63, 49, 24] can significantly restrict an attacker's ability to launch code reuse attacks. Unfortunately, CFI-permitted gadgets may still be enough to achieve DEP bypass and launch a code injection attack [25, 32, 31, 22]. Even fine-grained CFI approaches [46, 57, 43, 58] have been shown to be bypassable [28].

A substantial fraction of the defense techniques described above, including all bounds-checking techniques, and fine-grained randomization [21, 20, 23] and fine-grained CFI [46, 57, 43] operate only on source code. In particular, low-level code that relies on inline assembly will not be protected. 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. In contrast, *CFCI's protection extends to all code, regardless of the language in which it is written, or the compiler used to compile it.*

Code integrity. Nanda et. al. [41] proposes an approach to detect foreign code in Windows. They enforce a code loading policy inside the kernel. Their implementation is based on BIRD [40]. Similarly, Seshadri et. al. [50] proposes a code loading policy for OS kernel. Their policy is enforced by a tiny hypervisor.

MIP [45] and MCFI [46] also incorporate some code integrity features, implemented using policy checks on `mmap` and `mprotect`. However, their policy is weak in that it only aims to ensure existing code is never writable and executable. Moreover, the policy is insufficient to overcome challenges of dynamic loader and code patching. Attackers can divert control flow to dynamic loader or loader code statically linked into a code module to bypass their policy.

Securing loaders. Realizing the importance of preventing abuse of code loading privileges, Payer et.al. [48] developed TRuE, a system that replaces the standard loader with their secure version. The need to replace the system loader poses challenges for 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 [3, 12, 13, 4, 5, 9]. Thus, trusting the complete loader codebase for code integrity leads to a large trusted computing base (TCB). In contrast, CFCI implementation uses a very small reference monitor whose size is no more than 300 lines, including C and x86 assembly code. In comparison, a typical dynamic loader is 28KLoC.

7. CONCLUSIONS

In this paper, we presented an effective countermeasure against the threat of ROP attacks. Our 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. Our defense combines coarse-grained control-flow integrity with a comprehensive defense against native code injection in order to defeat these attacks. Our approach tracks 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 our system, CFCI. A key benefit of our 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. CFCI introduces almost no additional overheads at runtime over CFI, making it a promising candidate for deployment.

8. REFERENCES

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

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

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

[4] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[17] 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.

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

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

[20] S. Bhatkar and R. Sekar. Data space randomization. In *DIMVA*, 2008.

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

[22] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security*, 2014.

[23] 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 *S&P*, 2015.

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

[25] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose.

Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[26] 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.

[27] T. Durden. Bypassing pax aslr protection. Technical report, Phrack Magazine, vol. 0x0b, no. 0x3b, 2002.

[28] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *CCS*, 2015.

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

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

[31] 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.

[32] E. GÃűkta, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *S&P*, 2014.

[33] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *ACM CGO*, 2012.

[34] R. W. M. Jones, P. H. J. Kelly, M. C, and U. Errors. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG*, 1997.

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

[36] L. Li, J. E. Just, and R. Sekar. Address-space randomization for windows systems. In *ACSAC*, 2006.

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

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

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

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

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

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

[43] B. Niu and T. Gang. Per-input control-flow integrity. In *CCS*, 2015.

[44] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity.

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

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

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

[48] M. Payer, T. Hartmann, and T. R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *S&P*, 2012.

[49] J. Pewny and T. Holz. Control-flow Restrictor: Compiler-based CFI for iOS. In *ACSAC*, 2013.

[50] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP*, 2007.

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

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

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

[54] 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 *S&P*, 2013.

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

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

[57] 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.

[58] V. van der Veen, D. Andriesse, E. Goktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *CCS*, 2015.

[59] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. 2004.

[60] 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 *S&P*, 2009.

[61] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *ACM ASIACCS*, 2010.

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

[63] 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 *S&P*, 2013.

[64] M. Zhang. PSI: Platform for Static binary Instrumentation. http://seclab.cs.sunysb.edu/seclab/download.html.

[65] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *ACM VEE*, 2014.

[66] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, 2013.

[67] M. Zhang and R. Sekar. Squeezing the dynamic loader for fun and profit. http://seclab.cs.sunysb.edu/seclab/pubs/seclab15-12.pdf, 2015.