Fine-Grained Control-Flow Integrity through Binary Hardening

Mathias Payer[1], Antonio Barresi[2], and Thomas R. Gross[2]

[1] Purdue University
[2] ETH Zurich

Abstract. Applications written in low-level languages without type or memory safety are prone to memory corruption. Attackers gain code execution capabilities through memory corruption despite all currently deployed defenses. Control-Flow Integrity (CFI) is a promising security property that restricts indirect control-flow transfers to a static set of well-known locations.

We present Lockdown, a modular, fine-grained CFI policy that protects binary-only applications and libraries without requiring source-code. Lockdown adaptively discovers the control-flow graph of a running process based on the executed code. The sandbox component of Lockdown restricts interactions between different shared objects to imported and exported functions by enforcing fine-grained CFI checks using information from a trusted dynamic loader. A shadow stack enforces precise integrity for function returns. Our prototype implementation shows that Lockdown results in low performance overhead and a security analysis discusses any remaining gadgets.

1 Introduction

Memory corruption vulnerabilities are still one of the most critical types of bugs found in modern software systems. The majority of code running on current systems is written in C or C++. It is simply impossible to rewrite all these applications in a memory safe language due to the large amount of existing code. In addition, the problem of memory corruption is not restricted to low-level languages as safe languages are often implemented using low-level languages (e.g., the HotSpot Java virtual machine is implemented in C++, the CPython Python runtime is implemented in C, and Perl is implemented in C) or use native runtime libraries. Since 2006, a number of defense mechanisms like Address Space Layout Randomization (ASLR) [35], Non-Executable Memory (W$\oplus$X)/Data Execution Prevention (DEP) [46], stack canaries [18], and safe exception handlers have been deployed in practice to limit the power of attacker-controlled memory corruption. Unfortunately, all commonly deployed defense mechanisms can be circumvented as shown by current control-flow hijack attacks.

Control-Flow Integrity (CFI) [1,4,12,16,27,33,34,38,45,47,49–52] is a security property that restricts the set of targets that can be reached by any control-flow transfer to a statically determined control-flow graph. Current implementations share one or more drawbacks: (i) binary-only approaches [50–52] are restricted in their precision due to an over-approximation of the target sets where too many targets are allowed (these coarse-grained CFI policies can be exploited by attackers [8,14,17]), (ii) the need to recompile applications [4,16,34,38,45,47,49], (iii) no support (or protection) for shared libraries [1,4,16,33,38,47], or (iv) no stack integrity protection [12,34,45,48,51,52]. Modular CFI (MCFI) [34] recently added support for shared libraries but a recompilation of the application and all libraries is required. Furthermore, MCFI might require source code changes in the application, comes with its own `libc` implementation, which makes it less flexible, and does not support C++ code. COOP [40] presents an attack against CFI mechanisms that are unaware of C++ semantics for virtual function calls. Both MCFI and COOP were developed concurrently with Lockdown.

This paper presents Lockdown, a modular fine-grained CFI policy that supports any legacy binary. All indirect control-flow transfers are instrumented with security checks through dynamic binary translation. The target-sets for fine-grained CFI are approximated based on import and export definitions used in applications and libraries and a dynamic on-the-fly binary analysis for addresses of function pointers that are taken during the execution of the application. Using this approach, Lockdown adjusts the control-flow graph at runtime as code is being executed, growing the CFG when new code is executed, shrinking the CFG when libraries are unloaded. To protect against all forms of Return-Oriented Programming (ROP), Lockdown employs a shadow stack that enforces precise integrity of return instruction pointers. A shadow stack is stricter than a CFI check as the CFI check would allow the return instruction to target any possible call site of the current function while the shadow stack only allows the return to target the actual caller (by keeping state through the call/return relationship). A prototype implementation of our fine-grained CFI policy results in 19% average performance overhead for SPEC CPU2006 (which is on the same order as fine-grained source-level CFI implementations) and the performance overhead for Apache 2.2 is between 1.83% and 7.87% (depending on the configuration). This paper makes the following contributions:

1. Demonstration that CFI handles dynamic loading without recompilation; Lockdown is the first binary-only, modular, fine-grained CFI solution that supports dynamic loading and prevents real-life control-flow hijack attacks.
2. A performance evaluation of Lockdown. Lockdown results in low performance overhead of 19.09% on average for SPEC CPU2006 and between 1.83% and 7.86% on average for different Apache 2.2 configurations.
3. A CFI effectiveness and security evaluation of our fine-grained CFI approach going beyond the traditional quantitative methods. Our qualitative security evaluation method can be used to evaluate future CFI implementations.

2 Attack Model

Lockdown protects applications against control-flow hijack attacks under a powerful attack model: an attacker has read and write capabilities of the program's data regions and read capabilities of the program's code regions. This attack model reflects common efforts to circumvent deployed defenses on modern systems. The attacker uses memory corruption vulnerabilities present in the application or any of the loaded libraries to modify the program's data, thereby affecting the control-flow and execution of the program.

The attacker can neither modify the code region of the program nor inject additional code into the program. This assumption is fulfilled by current systems that enforce a W⊕X [46] strategy, where any memory area is either writable or executable (and never both at the same time). Also, the attacker cannot forcefully load attacker-controlled libraries. To achieve code execution capabilities the attacker must therefore reuse existing code sequences available in some code region of the program or its libraries [3,5,6,9,31,39,43]. As with any other CFI defense mechanism, non-control data attacks [10] are out of scope.

Using these given (practical) capabilities an attacker will try to (i) overwrite a code pointer, (ii) prepare a set of invocation frames for a code-reuse attack, and (iii) force the program to dereference and follow the compromised code pointer to achieve code execution.

3 Background and Related Work

A variety of defense mechanisms exist that protect against control-flow hijack attacks (see [44] for a systematization of attacks and defense mechanisms). Existing defense mechanisms stop control-flow hijack attacks at different stages by: (i) retrofitting type/memory safety onto existing languages [20,30], (ii) protecting the integrity of code pointers (i.e., allowing only valid code locations to change the memory area of a code pointer) [2,22,28,29], (iii) randomizing the location of code regions or code blocks (ASLR or code diversification are examples of probabilistic protections) [11,18,23,35], or (iv) verifying the correctness of code pointers when they are used, e.g., Control-Flow Integrity (CFI) [1]. CFI does not prevent memory corruption but detects corrupted values when they are used in indirect control-flow transfers. Unfortunately, CFI has not yet seen widespread use as existing implementations either affect the software development process (for source-based, fine-grained policies) or were shown to be insecure.

3.1 Control-Flow Integrity

Control-Flow Integrity (CFI) [1] and its extension XFI [16] restrict the control-flow of an application to a statically computed control-flow graph. Each indirect control-flow transfer (an indirect call, indirect jump, or function return) is allowed to transfer control at runtime only to the set of statically determined targets of this code location.

CFI relies on code integrity (i.e., an attacker cannot change the executed code of the application). Under this assumption, an attacker can only achieve code execution by controlling code pointers. CFI checks the integrity of code pointers at the location where they are used in the code. Using memory corruption vulnerabilities, an attacker may change the values of code pointers (or any other data). The attack is detected (and stopped) when the program tries to follow a compromised code pointer that refers to a location that is not in the set of allowed targets for a specific control-flow transfer instruction.

The effectiveness of CFI relies on two components: (i) the (static) precision of the control-flow graph that determines the upper bound of precision, and (ii) the (dynamic) precision of the individual runtime checks. First, CFI can only be as precise as the control-flow graph that is enforced. If the control-flow graph is too permissive, it may allow illegal control transfers. All existing CFI approaches rely on two phases: an explicit static analysis phase and an enforcement phase that executes additional checks. Most compiler-based implementations of CFI [1, 2, 4, 16, 33, 38, 47, 49] rely on a points-to analysis for code pointers at locations in the code that execute indirect control-flow transfers. A severe limitation of these approaches is that all the protected code must be present during compilation as they do not support modularity or shared libraries. Implementations based on static binary analysis [13, 27, 48, 50, 51] either rely on relocation information (e.g., in the Windows PE executable format) or reconstruct that information using static analysis [27, 52]. MCFI [34] is a recent compiler-based CFI tool that stores type information and dynamically merges points-to sets when new libraries are loaded (but does not support library unloading). Second, the initial upper bound for precision is possibly limited through the implementation of the control-flow checks (see [17] for common limitations in CFI implementations). Coarse-grained policies maintain few global sets of possible targets instead of one set per control-flow transfer: one target set each for indirect jumps, indirect calls, and function returns. The control-flow checks restrict the transfers to addresses in each set. This policy is an improvement compared to unchecked control-flow transfers but overly permissive as an attacker can hijack control-flow to any entry in the set. As demonstrated in recent work [8, 14, 17] these coarse-grained CFI implementations still allow attackers to successfully mount code-reuse attacks by making the exploits CFI aware, i.e., hardening them to just use gadgets still allowed in the limited set of valid targets. COOP [40], developed concurrently with our work, presents such an attack, leveraging virtual calls in C++ programs to circumvent CFI mechanisms that do not restrict call targets for virtual calls based on a type-based analysis. Recovering such precise information is hard (and often infeasible) for a binary analysis. The precision of the Lockdown CFI policy relies on the shared libraries that are used. If the same amount of code is broken into smaller libraries then the precision of Lockdown increases, limiting attacks like COOP to call targets that are imported in the current object.

Lockdown is a dynamic fine-grained CFI approach for unmodified binary-only applications (i.e., no source access is needed) that enforces a stricter, dynamically constructed *modular* control-flow graph using CFI checks and stack integrity using a shadow stack, than obtained in earlier approaches. Lockdown ensures code integrity, adds a shadow stack that protects against ROP attacks, and enforces dynamic control-flow checks for all indirect control-flow transfers.

3.2 Dynamic Binary Translation

Software-based Fault Isolation (SFI) protects the integrity of the system and/or data by executing additional guards that are not part of the original code. Dynamic Binary Translation (DBT) allows the implementation of SFI guards on applications without prior compiler involvement by translating code on the fly.

The DBT system can dynamically enforce security policies by collecting runtime information and restricting capabilities of the executed code [21].

Several DBT systems exist with different performance characteristics. Valgrind [32] and PIN [25] offer a high-level runtime interface resulting in higher performance costs while DynamoRIO [7] and libdetox [36] support a more direct translation mechanism with low overhead, translating application code on the granularity of basic blocks. Lockdown builds on libdetox, which has already been used to implement several security policies.

A security policy can only be enforced if the translation system itself is secure. Libdetox splits the user-space address space into two domains: the untrusted application domain and the trusted and protected binary translator domain. This design protects the binary translation system against memory corruption attacks. Libdetox uses a separate translator stack and separate memory regions from the running application. Libdetox protects the trusted domain by randomizing address locations[3] and enforces the following properties: (i) all code is translated before execution; (ii) translated code can only access the application domain; (iii) no pointer to the trusted domain is ever stored in the application domain, protecting the trusted domain against information leakage. The application traps into the trusted domain when (i) it executes a system call, (ii) reaches untranslated code, or (iii) a full (non-inlined) security check is triggered. Libdetox relies on a trusted loader [37], protecting the DBT system from attacks against the loader when loading or unloading shared libraries.

Lockdown gives the following security guarantees: a *shadow stack* protects the integrity of return instruction pointers on the stack at all times; the *trusted loader* protects the data structures that are used to execute functions in other loaded libraries at runtime; and the *integrity* of the security mechanism is guaranteed by the binary translation system. The shadow stack is implemented by translating call and return instructions [36]. Translated call instructions push the return instruction pointer on both the application stack and the shadow stack in the trusted domain. Translated return instructions check the equivalence between the return instruction pointer on the application stack and the shadow stack; if the pointers are equivalent then control is transferred to the translated code block identified by the code pointer on the shadow stack. The existing version of libdetox supports the full x86 instruction set (including SSE extensions).

4 Lockdown Design

Lockdown enforces a fine-grained, dynamic, modular CFI policy at the granularity of individual Dynamic Shared Objects (DSO) (a DSO is an individual ELF file like the program or any used shared library) and symbols (applications or libraries for calls, symbol definitions for jumps). Lockdown restricts (i) inter-module indirect calls to functions that are exported from one object and

[3] The trusted domain is small both regarding code and data. An alternative implementation uses SFI and mask operations (added by the binary translator to any read-/write in the application domain) to protect against information side channels [42], resulting in higher overhead.

imported in the other object, (ii) intra-module indirect calls to valid functions, (iii) indirect jump instructions to valid instructions in the same function and valid call targets for tail calls, and (iv) return instructions to the precise return address (with a special handler for exceptions). Figure 1 and Figure 2 show examples for call and jump restrictions. The per-object target sets are adapted dynamically whenever libraries are loaded or unloaded. The integrity of return instructions is enforced at all times using a shadow stack. Indirect call instructions and indirect jump instructions execute a runtime check that validates the current target according to the current location and the object's sets.

The ELF format [15,41] specifies the on-disk layout of applications, libraries, and compiled objects. ELF files contain symbol information with different level of detail. The ELF section `dynsym` provides the required symbol information for inter-module calls and the optional `symtab` ELF section contains information about all symbols. Lockdown uses both symbol tables to construct the per-object target sets (and to infer function boundaries to restrict jumps). If only the coarse-grained `dynsym` table is available, e.g., because the library is stripped,

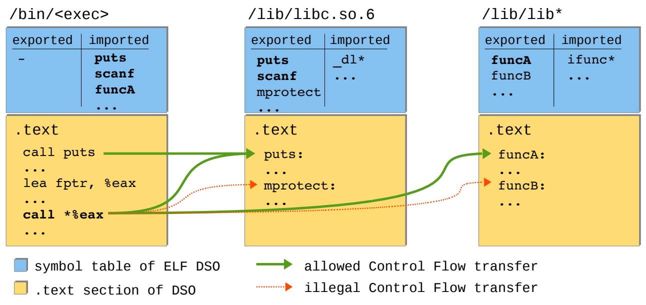


Fig. 1. Call restrictions for an executable and two libraries. DSOs are only allowed to call imported function symbols. Local function calls may only transfer to local function symbols.

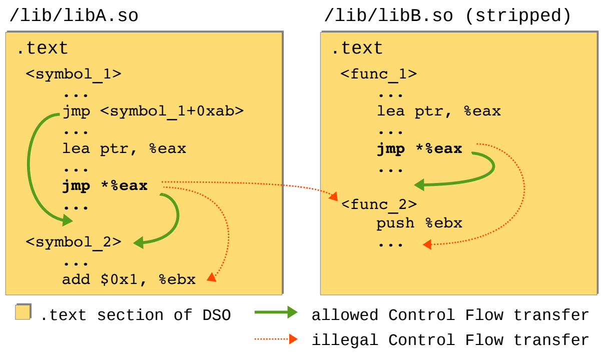


Fig. 2. Restrictions for jump instructions within two different libraries. Jumps may only target locations within the same symbol. Inter-DSO jumps are not allowed in general. If stripped, jumps have to stay within the bounds of the nearest symbol definitions.

then Lockdown falls back to a coarser-grained protection for intra-object calls and jumps. The policies for inter-object transfers and return targets are not affected. On current Ubuntu systems (and other popular Linux distributions) the full `symtab` information is available and easily installable for user-space applications and all common libraries.

In contrast to other binary CFI solutions where only one global static set of valid targets per indirect control-flow type exists (e.g., one global set for all return instructions), Lockdown provides a finer-grained approach where the set of valid targets is dynamic and specific to the instruction type, its object (for calls and jumps), its symbol within the object (for jumps), and the currently active stack frames at the time of the control-flow transfer (for returns). This setup results in a fine-grained CFI policy compared to other coarse-grained approaches that solely use (few) global sets for all valid targets. The precision of the Lockdown CFI mechanism depends on the modularity of the application. If the same amount of code is split across more libraries (e.g., one library is split into multiple libraries) then the precision of our CFI mechanism increases and becomes finer-grained. Modern applications are often implemented in a highly modular way with, e.g., LibreOffice using 297 libraries, Chromium-browser using 194 libraries, or Firefox using 29 libraries.

4.1 Rules for Control Transfers

Three different forms of indirect control transfers exist, `call`, `jump`, and `return` instructions. The rules for different control transfers are as follows:

1. Call instructions must always target valid functions. The set of valid functions is specific for each object. Only functions that are defined or imported in the current object are allowed as targets;
2. The target of a jump instruction must (i) target a valid instruction inside the current function (according to a linear disassembly from function beginning) or (ii) target a valid function target (due to tail call optimizations);
3. Return instructions must always transfer control back to the caller (or to one of the previous callers in exceptional cases). The DBT system transparently keeps a shadow stack data structure in the trusted domain (not visible to the application) to verify return addresses on the stack before they are dereferenced. Therefore, Lockdown always enforces the precise return target.

Additionally, Lockdown implements a set of special handlers for control-flow particularities in low-level libraries (e.g., libc) as discussed in Section 5.2.

4.2 Control Transfer Guards

Generally, the target of an indirect control-flow transfer is not known in advance. Therefore, a runtime check is needed and the binary translator emits a guard that executes these dynamic checks when an indirect control-flow transfer executes. Conceptually, each indirect control-flow transfer traps into the Lockdown domain and verifies that the current target is allowed according to the current location and the rules defined in Section 4.1. To avoid unnecessary resolutions, different optimizations are implemented. For each DSO a lookup table is used

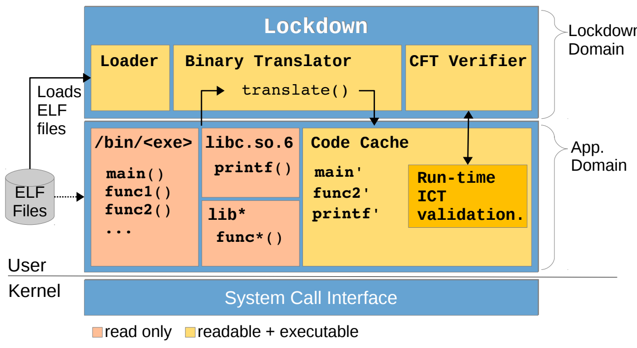


Fig. 3. Overview of the Lockdown approach. (CFT: Control-Flow Transfer, ICT: Indirect Control-Flow Transfer, ELF: Executable and Linkable Format)

to cache already verified transitions that can be used in the fast path. Further optimizations are described in Section 5.1.

Return instructions are translated to check the shadow stack. The added guard verifies that the current return address on the application stack is the same as the address that was pushed by the last call instruction. The shadow stack in the Lockdown domain keeps track of valid return targets. To allow different exceptional cases (e.g., tail calls, or C++ exception handling) the shadow stack can be resynchronized with the application stack by unwinding shadow stack frames until the frames match again. Shadow stack frames can only be *removed* by exceptions (never added), resulting in sound behavior.

4.3 Handling Stripped Binaries

For many systems, full symbol information is available (e.g., the symbol tables of all common libraries for Ubuntu 12.04 are available as separate packages). Yet, some binaries are only available in stripped form. Even if binaries are stripped, imported and exported symbols are always available and Lockdown enforces a policy that is fine-grained even without full symbol tables. The same strong CFI policy is enforced for all inter-module transfers as all required information remains available. For intra-module transfers we currently rely on heuristics (see Section 5.3) that detect valid transfers. A more detailed (static or dynamic) binary analysis could recover function prologues more precisely, increasing the intra-module protection to the level where full symbol information is available.

5 Prototype Implementation

The prototype implementation (Figure 3) builds on libdetox and secuLoader [36, 37]. The DBT embeds the control-flow checks during the translation of individual basic blocks. All static control-flow transfers are verified during translation and indirect (dynamic) control-flow transfers are instrumented to execute an inlined dynamic guard depending on the control-flow transfer type.

For all DBT systems, one of the biggest performance overheads is the translation of indirect control-flow transfers. Such transfers cannot be translated ahead of time and always incur a runtime lookup to consult the mapping between the original code and the translated code blocks before transferring control-flow to the translated code. Lockdown executes dynamic CFI checks right before the runtime lookup. To reduce the overhead of these indirect control-flow transfers Lockdown implements a set of optimizations that cache data at different levels (e.g., a cache for each indirect control-flow transfer location, a cache for lookup values in the mapping table, and a cache for frequently used symbols).

The prototype is released as open-source and is implemented in less than 24k loc (C code mixed with a small amount of inline assembly): 2k loc for Lockdown, 15k loc for the binary translator and 7k loc for the trusted loader.

5.1 Runtime Optimizations

Achieving low overhead when running binary-only applications is a challenging problem: to support dynamic CFI policies, Lockdown needs to run the binary analysis alongside the executing application. We have implemented a set of optimizations to achieve low overhead without loss of precision or security. The libdetox DBT engine already implements a set of optimizations like local inline caches for indirect control-flow transfers. We extend these lookups by a control-flow transfer check. The validation uses a lookup table for already verified {`source, destination`} pairs and recovers if this fast check fails (e.g., on a hash miss in the lookup table).

Furthermore, Lockdown uses inlined optimizations for different guards to avoid the validation path whenever possible. To understand indirect control-flow behavior we measured a large set of applications. Out of all indirect control-flow transfers, 51.04% are intra-DSO calls, 19.88% are inter-DSO calls, 28.99% are intra-DSO jumps with only 0.09% inter-DSO jumps. Inlined CFI checks for indirect jumps (i) check if the target stays inside the same symbol, (ii) execute a fast, direct lookup for intra-DSO targets, or (iii) redirect to a slow path. Inlined CFI checks for indirect calls (i) consult a local cache if the current target equals the last target, or (ii) redirect to the slow path that executes a full validation and updates the local cache as a side effect. Lockdown calculates symbol boundaries at translation time. Correctness of inlined checks follows due to immutability of symbol definitions and code (given by W$\oplus$X). These optimizations effectively avoid the slow path (and full table lookup) and therefore reduce the performance overhead introduced by the control-flow transfer checks significantly.

5.2 Control-Flow Particularities

Control-flow transfers in off-the-shelf binaries do not always adhere to the rules listed in Section 4.1 and Lockdown catches this behavior and recovers using a set of handlers. ELF implements calls to other libraries as a call to the PLT section of the current module and an indirect jump to the real function [15]. As Lockdown can only rely on binaries (a source code approach could implement this differently) it replaces such calls with a direct call to the loaded function, removing the indirect jump and needed CFI guard. Some special cases are specific to low-level libraries like the libc runtime support functions, e.g., inter-module calls to symbols that were not imported, intra-module cross function jumptables, inter-module callback functions, or even inter-module calls targeting PLT entries which would bypass our PLT inlining if not handled correctly. Lockdown also allows indirect jumps as tail calls to the beginning of other functions in the set of currently allowed call targets. Although a variety of these special cases exist, our experience shows that they can all be handled by a small set of handlers without compromising the CFI security properties.

High-level application code (i.e., all code that is not the libc or other low-level functionality like the loader) adheres to the rules listed in Section 4.1 and therefore does not require special handling, except callback functions (function pointers), which are discussed in the next section.

5.3 Implementation Heuristics

Binaries have little information about the types that are used at runtime and it is not always possible to recover information precisely. To support callback functions (i.e., a function in a library returns a function pointer that is later called from a different library; if this function is not exported/imported then the CFI guard would fail), Lockdown implements a dynamic scanning technique that is similar in design to the static analysis of Zhang and Sekar proposed in [52].

Lockdown uses the following patterns to detect pointers to callback functions on the fly (i) `push imm32`, where a function pointer is pushed onto the stack, (ii) `movl imm32, rel(%esp)`, where rel references a local variable on the stack, and (iii) `leal imm32(%ebx), %e*x`, where a function pointer is moved from memory into a general purpose register relative to GOT.PLT [15], or (iv) relocations that are used to define pointers for many callbacks (e.g., `R_386_RELATIVE`). In addition, Lockdown scans data sections (`.data` and `.rodata`) to detect static code pointers. Lockdown's dynamic analysis allows us to use the actual values and hard-code references in our guards. Each code pointer is verified to target a valid code location. These heuristics detect code pointers at the source where they are either encoded in instructions or stored in read-only data. Such heuristics will miss pointers that are modified using pointer arithmetic or taken from attacker-controlled, writable memory. We verified that these heuristics are sufficient to protect a large set of common Unix applications. To prevent attacker-controlled manipulation of our pointer detection heuristics we ensure that code pointers detected by heuristics can only come from read-only data or the trusted loader (e.g., through the GOT.PLT section). Our heuristics purposely only detect *few* targets that can be called from any module. Note that using heuristics to find *all* possible targets would degenerate to a weak CFI policy that is open to existing attacks. A finer-grained attribution of targets to modules through a data-flow analysis remains an interesting research question.

5.4 Binary Compatibility

Binary compatibility is a challenge for approaches like Lockdown, and using heuristics leads to a formally unsound approach. We successfully tested our implementation with a large set of applications, compilers, and optimization levels without problems. Our experience shows that low-level particularities (e.g., of the libc) can be handled in an automated and efficient way. As a dynamic approach, there are few limits to implement special handling and therefore Lockdown can easily be extended to account for possibly remaining special cases where binary compatibility is broken. An example concern might be that `ret` instructions are implemented as `pop;jmp*;` sequences. We did not experience this potential issue in any of our tests, but such sequences can be handled by simulating a `ret` instruction. Furthermore, all recently proposed CFI approaches [34, 50–52] come with either a binary or source code compatibility risk and none of them can guarantee full compatibility. Even MCFI [34] cannot guarantee full C source code compatibility and requires source code changes within the SPEC CPU2006 benchmark suite. Under Lockdown, no binary compatibility issues where encountered for tested applications (including SPEC CPU2006), and no source-code changes were necessary.

The current prototype supports arbitrary user-space x86 code, shared libraries, signals, setjmp/longjmp, and multi-threaded applications. Self-modifying code or dynamic code generation is not supported. This is not a general limitation as a dynamic approach like Lockdown can detect runtime code generation and translate this new code alongside regular application code. A challenge for the binary translator is to detect if the dynamic code was generated by the benign application or by the attacker.

6 Evaluation

We evaluate Lockdown in the following areas: (i) performance using the SPEC CPU2006 benchmarks, (ii) real-world performance using Apache 2.2, (iii) a discussion of the security guarantees according to the implemented security policy, and (iv) remaining attack surface.

We run the experiments on an Intel Core i7 CPU 920@2.67GHz with 12GiB memory on Ubuntu Linux 12.04.4 LTS (on 32-bit x86). Lockdown and the SPEC CPU2006 benchmarks are compiled with GCC version 4.6.3 and -O2 optimization level. The full set of security features of Ubuntu Linux 12.04.4 LTS is enabled (ASLR, W$\oplus$X, stack canaries, and safe exception frames).

6.1 Performance

We use the SPEC CPU2006 benchmarks to measure CPU performance and to evaluate the performance impact of (i) Lockdown and (ii) binary translation only, both compared to native execution, in Figure 4. Due to issues with the trusted loader [37] combined with new versions of the libc we were unable to run omnetpp and dealII. The binary translation column includes the overhead from the trusted loader. The additional average overhead introduced by Lockdown for CFI enforcement (compared to binary translation) is 4.45%, and the average overhead for Lockdown in total is 19.09%. Only five benchmarks have a total overhead of more than 45% if run under Lockdown. The majority of benchmarks face a reasonable performance overhead of less than 10%.

An average overhead of 19.09% may seem high, but we point out that Lockdown enforces a purely dynamic binary-only fine-grained CFI policy that runs the binary analysis alongside the executed application whilst supporting dynamic library loading. Other fine-grained compiler-based CFI policies report comparable overhead of 20% overhead for SPEC CPU2006 [49] . MCFI, a compiler-based solution, recently reduced this overhead to 5-6% for a subset of SPEC CPU2006

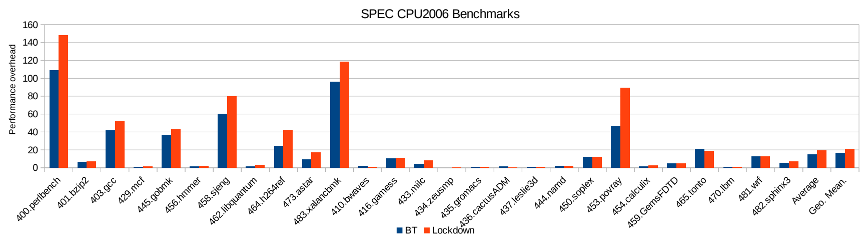


Fig. 4. SPEC CPU2006 overhead for binary translation only and Lockdown (full CFI protection including shadow stack).

benchmarks [34] without enforcing stack pointer integrity (which is essential to protect against ROP attacks).

A dynamic, binary-only, fine-grained CFI policy faces several challenges: (i) type information must be recovered using binary analysis and (ii) the analysis must be low-overhead and carried out alongside the executed application. The overhead of Lockdown currently includes 14.64% for the shadow stack, trusted loader, and binary translation as well as 4.45% for the CFI target analysis and dynamic checks. A combination of static and dynamic binary rewriting may reduce the performance impact further.

6.2 Apache case study

We evaluate the performance of a full Apache 2.2 setup running under Lockdown. Apache is set up in the default configuration. To test the performance of the web server we use an HTML file (56 KB) and a jpg image (1054 KB). The file sizes correspond to average HTML and image sizes [19]. We used `ab` (Apache benchmark tool) to send 15,000,000 requests for each file in 3 configurations (single threaded, 10 concurrent connections, 10 concurrent connections with keep-alive) and measured the overall time required to respond to these requests.

Table 1 shows the overhead of running Apache 2.2 under the Lockdown CFI policy. The overhead for smaller files is generally higher due to the additional context switches between translator domain and application domain for file and network operations. Apache sends files using as few I/O operations as possible and with small files there is not enough computation that is executed to recover from the performance hit of the context switch. In the single-threaded configuration the overhead is high compared to the concurrent configurations due to additional translation and lookup overhead as threads are not reused as many times as for the concurrent configuration. This case-study shows that the overhead for Lockdown is small in real-world contexts.

Configuration	Small file	Image	Combined
Single threaded	30.41%	1.94%	7.87%
Concurrent	6.27%	1.09%	1.83%
Concurrent with keep-alive	15.80%	3.00%	4.36%

Table 1. Apache 2.2 benchmark results.

6.3 Security and CFI Effectiveness Case-Study

Evaluating the effectiveness of a CFI implementation in terms of security is non-trivial. Running a vulnerable program with a CFI mechanism and preventing exploitation using one specific vulnerability does not guarantee that the vulnerability is not exploitable under other circumstances (hijacking a different indirect branch instruction, overwriting other control-flow sensitive data, or just using different gadgets in a code-reuse attack).

We make the following observations: (i) in our attack model a successful attacker needs to hijack the control-flow to already executable code within the process, (ii) the probability of success for an attacker depends on the ability to find a sequence of reusable code (gadgets) that (executed in the right order) accomplishes the intended malicious behavior (e.g., spawning a shell).

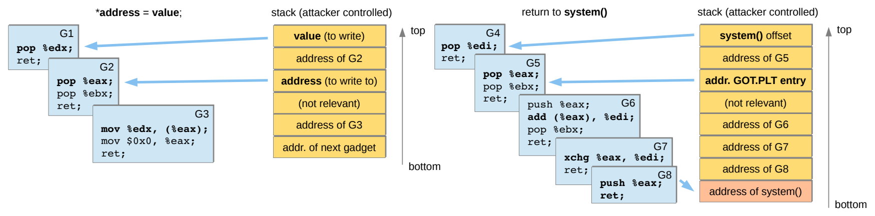
The effectiveness of a CFI implementation therefore depends on how effectively an attacker is restrained in the ability to find and reuse already available code. This directly translates to the *quantity* and *quality* of the remaining indirect-control flow targets of the enforced CFG.

CFI effectiveness with AIR Zhang and Sekar [52] propose a metric for measuring CFI strength called Average Indirect target Reduction (AIR). AIR exclusively focuses on the quantity of remaining gadgets ignoring quality and therefore fails to capture the effectiveness of CFI policies regarding security. This limitation is underlined in recent work [8, 14, 17] where coarse-grained CFI implementations proved to be ineffective while having very high AIR values (>99%).

For a program like LibreOffice, which maps at least 56,417,429[4] bytes of executable memory, for a CFI solution with AIR of 99% the remaining 1% of valid targets still consist of 564,174 potential targets. An attacker needs to find only a handful of gadgets within this set of valid targets to successfully exploit a vulnerability. Hence, there is no AIR threshold that indicates if a CFI policy is secure or not; we present AIR numbers only to allow comparison with related work. Lockdown enforces a dynamic policy which extends and shrinks target sets depending on the executed code, we therefore report numbers at the end of execution. Lockdown achieves an AIR value of *99.88%* for SPEC CPU2006 (*99.84%* with stripped libraries) and 99.55% for an application set of LibreOffice, Apache, Vim, and xterm. In comparison, static CFI implementations have AIR values for SPEC CPU2006 of 99.13% (CFI reloc) or 98.86% (CFI bin) [52].

CFI Security Effectiveness Case Study We provide an in-depth analysis of the effectiveness of Lockdown by looking at CVE 2013-2028, a memory corruption vulnerability for nginx. Modern attacks exploiting memory corruption vulnerabilities rely on code-reuse techniques. The widespread deployment of W$\oplus$X does not allow to execute memory areas like the stack or the heap. A code-reuse attack [3, 5, 6, 9, 31, 39, 43] combines already executable code snippets (gadgets) found in the executable area of a program or libraries to realise attacker desired behavior. Return-Oriented Programming (ROP) uses gadgets

[4] We looked at LibreOffice 3.5 on Ubuntu Linux 12.04.4 LTS and added all the initial executable ELF segments, i.e., of the soffice.bin and all its library dependencies.



```
G1
pop %edx;
ret;

G2
pop %eax;
pop %ebx;
ret;

G3
mov %edx, (%eax);
mov $0x0, %eax;
ret;

G4
pop %edi;
ret;

G5
pop %eax;
pop %ebx;
ret;

G6
push %eax;
add (%eax), %edi;
pop %ebx;
ret;

G7
xchg %eax, %edi;
ret;

G8
push %eax;
ret;
```

Fig. 5. Gadgets in our nginx 1.4.0.0 exploit (on Ubuntu Linux 12.04.4 LTS). G1-G3 implement a write primitive and G4-G8 transfer control to an arbitrary libc function.

ending with a return instruction while Jump-Oriented Programming (JOP) and Call-Oriented Programming (COP) use gadgets ending with indirect jumps and indirect calls. Most real-life code-reuse attacks rely on return gadgets.

CVE 2013-2028 reports a signedness bug in nginx before 1.4.0.0. The vulnerability can be exploited by overflowing the stack [26] or corrupting heap data [24]. ASLR and stack canaries are bypassed by server-side heap spraying and brute-forcing. The gadgets in Figure 5 can be used to exploit the vulnerability and to execute a remote bind shell. Gadgets 1-3 (G1-G3) are used to implement a "write value to address" primitive and gadgets 4-8 (G4-G8) are used to load a GOT.PLT entry into a register, add an offset to it, push it onto the stack and transfer control it. The "write value to address" primitive is used several times to copy a sequence of 4 byte values to a writeable memory area (somewhere within the .data section) and then gadget sequence 4-8 is used to perform an arbitrary function call. More specifically the copied data will be used as the function argument to libc's **system()** function. This technique allows an attacker to execute arbitrary commands (e.g., a remote shell).

An attacker needs only a small number of gadgets that, when chained together, allows the implementation of the desired behavior. To assess the security effectiveness of a CFI implementation a detailed *application specific* analysis is required. We implemented a gadget finder tool that identifies potential gadgets within the set of valid target locations for a particular indirect control-flow transfer, returning all potential gadgets (according to additional filter criteria).

A filter criterion for gadgets is the type of indirect branch instruction at the end of the gadget: return (ret), indirect jump (jmp), or indirect call (call). We consider gadgets up to a length of 30 instructions. In practice, gadgets are short (up to 5 instructions). Longer gadgets might be used to bypass ROP heuristics [14, 17] but are harder to control due to unwanted side effects. We allow conditional and unconditional jumps within gadgets. Jumps to (i) locations outside of the gadget, (ii) illegal instructions, or (iii) instructions not allowed in user mode are not allowed in a gadget. We further introduce a *system gadget* (sys) type that ends with a system call (i.e. `int 0x80` or `sysenter`). These gadgets are used to construct syscall primitives.

To reliably perform JOP or COP attacks [5, 9] certain types of gadgets are needed. We follow the terminology from [5]. Gadgets are either initialization, dispatcher, or functional gadgets: An *initialization gadget* (init) contains a `popa` instruction or allows to pop or move at least three values from the stack into registers. Such gadgets allow the initialization of registers. Initialization gadgets in our evaluation can be return, jump, or call gadgets. *Dispatcher gadgets* (disp) are jump or call gadgets that change the target used for the indirect jump or call at the end of the gadget. Not all gadgets that fulfil this criterion are feasible dispatcher gadgets, providing an over-approximation of available gadgets. *Functional gadgets* (func) are the opposite of dispatcher gadgets, they do not change the content of the register later used for addressing of the indirect jump or call. The idea is to have a register holding the address of the dispatcher gadget so at the end of a functional gadget we can jump or call the dispatcher gadget to load and dispatch the next gadget. We refer to related work [5, 9] for details on gadget chain construction. Attacks that rely *exclusively* on JOP and COP (i.e., no return gadgets are used) need at least one initialization gadget (that initializes a set of registers from attacker supplied memory), one dispatcher gadget (that implements the 'load and branch' sequence to dispatch execution to the next gadget), and a set of functional gadgets (e.g., to move a value to a register).

We analyse remaining gadgets in nginx for (i) Lockdown and (ii) a coarse-grained CFI policy using the filtering described above. The coarse-grained CFI policy follows a conservative and strict static CFI policy where returns and jumps can target call sites (call preceded locations) and where calls are allowed to target only symbol addresses (beginning of functions). The coarse-grained CFI policy is stricter than the combined static CFI policy described in [14, 17]. Such an overly-strict CFI policy results in a lower bound of total targets that *must* be allowed by any coarse-grained CFI policy. Real policies must over-approximate this set of targets due to imprecisions in the analysis, therefore this policy is strictly stronger than any coarse-grained policy. We show that an attacker can achieve arbitrary computation for this overly strict policy that under-approximates the valid targets and thereby over-approximates the protection capabilities of all coarse-grained CFI policies, therefore all coarse-grained CFI policies are broken, generalizing prior attacks against coarse-grained CFI [8, 14, 17].

We provide a precise analysis of the remaining gadgets (which we inspected manually) for Lockdown and we show that no usable gadgets remain. Filtering the valid targets of the vulnerable 'ret' and 'call' instruction in nginx 1.4.0.0 under Lockdown (L_{ret}, L_{call}) and under the coarse-grained CFI policy (S_{ret}, S_{call}), results in the number of gadgets presented in Table 2. These are the only gadgets available for exploit construction.

Lockdown greatly reduces the available attack surface. For the vulnerable return instruction only 3 gadgets with a maximum size of 5 instructions are available (7 gadgets if longer sequences are considered). The set of valid targets only contains return gadgets, prohibiting a transition to JOP or COP. A further restriction is that the 7 gadgets can only be executed in a certain order, namely from top of the shadow stack down. Therefore, every gadget can only be executed once. Manual analysis of the specific instructions within the available gadgets shows limited computational abilities, making successful exploitation highly unlikely. We therefore conclude that Lockdown effectively prevents attacks targeting the vulnerable return instruction (for this vulnerability). The second possible exploitation vector of CVE 2013-2028 uses an indirect call instruction. Manual analysis of the gadgets available for the specific vulnerable indirect call instruction again shows limited computational abilities, making successful exploitation highly unlikely as well. First, only JOP-only or COP-only attacks are possible. Using return gadgets is no longer possible due to the strictness of Lockdown's policy for return instructions (the number of reachable return gadgets is shown in parentheses). For long gadgets we get few initialization or dispatcher gadgets but no functional gadgets. Longer gadgets come with an increasing risk of unwanted (and unrecoverable) side effects and/or loss of control-flow from the attacker's perspective. Even if one of the call gadgets could (hypothetically) be used as an arbitrary call, Lockdown always enforces a least-privileges policy for inter-DSO calls (if a symbol is not explicitly imported, the call is not allowed). Therefore even in the most extreme case where the redirection over a call gadget to a specific function would be sufficient, the called function must also be imported (for nginx, e.g., `system` or `mprotect` are not imported).

Looking at the results for the coarse-grained CFI policy shows that, despite the high AIR value, thousands of gadgets remain readily available. Most gadgets are return gadgets, allowing flexible ROP chains. In fact, we easily find a set of gadgets that implements the same primitives needed for successful exploitation even if the coarse-grained CFI policy is enforced. For the indirect call exploitation vector we simply fall back to ROP by using a stack pivot and one of the available return gadgets. Falling back to ROP is not possible for Lockdown because the shadow stack enforces stack integrity.

We conclude that Lockdown indeed prevents exploitation of CVE 2013-2028 for both exploitation vectors (while coarse-grained CFI fails to protect both exploitation vectors). Lockdown's strength originates from the combination of a precise policy for return instructions and a fine-grained policy for call/jump instructions. In contrast, our analysis for a representative coarse-grained CFI pol-

	length	total	ret	jmp	call	sys	init	disp	func	protected?
L_{ret}	5	3	3	0	0	0	0	0	0	✓
	10	6	6	0	0	0	1	0	0	✓
	15	7	7	0	0	0	1	0	0	✓
	30	7	7	0	0	0	1	0	0	✓
L_{call}	5	20 (97)	0 (77)	0	17	3	0	13	0	✓
	10	53 (270)	0 (217)	0	50	3	9	45	0	✓
	15	65 (392)	0 (327)	1	61	3	31	56	0	✓
	30	99 (586)	0 (487)	2	94	3	125	85	0	✓
S_{ret}	5	2037	1295	440	294	8	2	216	50	×
	10	3741	2662	536	533	10	261	326	69	×
	15	4622	3330	583	698	11	516	375	97	×
	30	6209	4450	763	980	16	1072	558	117	×
S_{call}	5	99	97	0	0	2	4	0	0	×
	10	401	391	0	4	6	9	4	0	×
	15	635	617	0	12	6	68	12	0	×
	30	954	922	0	24	8	268	24	0	×

Table 2. Gadgets found at valid return and call locations when protecting the vulnerable return and indirect call instruction in nginx 1.4.0.0 for Lockdown (L_{ret} and L_{call}) and the strict static CFI policy (S_{ret} and S_{call}).

icy shows that despite high AIR values such policies are unable to effectively prevent code-reuse attacks, further questioning the effectiveness of existing coarse-grained CFI techniques. In fact, current binary-only CFI implementations are even more permissive than the strict coarse-grained CFI policy discussed here. A common problem is the missing (strong) protection against ROP attacks.

Table 3 shows remaining gadgets under Lockdown for the vulnerable call instruction if *nginx and all libraries are stripped*. Due to the limited symbol information, more targets are reachable for the vulnerable call instruction compared to the unstripped binary. This leaves the attacker with several gadgets to choose from. As the shadow stack does not rely on symbol information, the precision for the stack remains the same as in Table 2 and the attacker must exclusively rely on COP or JOP. The majority of the available gadgets are intra-DSO targets as the information needed for inter-DSO control transfers is always available. While the attacker can construct Turing-complete computation inside one DSO, executing arbitrary system calls is only possible if the desired libc functions are imported into the DSO or an exploitable call or jump instruction inside libc is found. Therefore, even if binaries are stripped the Lockdown policy offers protection for transfers between objects and against ROP attacks. Here, an attacker is restricted to the few functions imported in the vulnerable object (therefore the protection is partial), e.g., in the case of libc as the destination DSO the list of allowed functions from a source DSO to libc is generally limited. If a security sensitive function (like `system()`) is still within the allowed targets then data validation guards could be emitted to detect attacks.

Our analysis shows that the combination of a strong policy for returns and a fine-grained CFI policy for indirect jumps and calls is key in preventing attacks. Missing only one of these fine-grained policies for either of the indirect branch instructions would already open up the attack surface for successful exploitation.

6.4 Security Guarantees

Lockdown enforces a strict, modular, fine-grained CFI policy for executed code combined with a precise shadow stack, resulting in the following guarantees: (i) the DBT always maintains control of the control-flow, (ii) only valid, legitimate instructions are executed, (iii) function returns cannot be redirected, mitigating ROP attacks, (iv) jump instructions can target only valid instructions in the same function or symbols in the same module (DSO), (v) call instructions can target only valid functions in the same module (DSO) or imported functions, (vi) all signals are caught by the DBT system, protecting from signal oriented programming, (vii) all system calls go through a system call policy check. Due to the modular implementation, individual guarantees build on each other: the

	length	total	ret	jmp	call	sys	init	disp	func	prot.?
L_{call}	5	4016 (20388)	0 (16372)	804	3203	9	2	563	403	(✓)
	10	6129 (34885)	0 (28756)	906	5206	17	1446	1323	565	(✓)
	15	6961 (45209)	0 (38248)	947	5990	24	3880	1707	618	(✓)
	30	10109 (64695)	0 (54586)	1017	9062	30	10932	3294	760	(✓)

Table 3. Gadgets found at valid call locations when protecting the vulnerable indirect call instruction in nginx 1.4.0.0 for Lockdown (L_{call}) *without debug symbol information.*

binary translator ensures the SFI properties that only valid instructions can be targeted, the shadow stack protects return instructions at all times, the trusted loader provides information about valid targets for `call` and `jmp` instructions, and the dynamic control-flow transfer checks enforce dynamic CFI.

7 Conclusion

This paper presents Lockdown, a fine-grained, modular, dynamic control-flow integrity policy for binaries. Using the symbol tables available in shared libraries and executables we build a control-flow graph on the granularity of shared objects. A dynamic binary translation based system enforces the integrity of control-flow transfers at all times according to this model. To counter recent attacks on coarse-grained CFI implementations, we use a shadow stack that protects from all ROP attacks.

Our prototype implementation shows low performance overhead of 19.09% on average for SPEC CPU2006. In addition, we reason about CFI effectiveness and the strength of Lockdown's dynamic CFI approach, which is more precise than other CFI solutions that rely on static binary rewriting. Our security evaluation goes beyond AIR metrics and we provide an in-depth analysis of our implementation's security effectiveness using real exploits that demonstrates a strong policy for returns must be combined with a fine-grained CFI policy for indirect jumps and calls if we want to prevent attacks.

Lockdown enforces strong security guarantees for current systems in a practical environment that allows dynamic code loading (of shared libraries), supports threads, and results in low overhead.

8 Acknowledgements

We thank Andreas Follner, Volodymyr Kuznetsov, Per Larsen, Kaveh Razavi, our shepherd Cristiano Giuffrida, and the anonymous reviewers for feedback and discussions. This research was supported, in part, by a grant from NSF.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: CCS'05 (2005)
2. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing memory error exploits with WIT. In: SP'08 (2008)
3. Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., Boneh, D.: Hacking blind. In: SP'14 (2014)
4. Bletsch, T., Jiang, X., Freeh, V.: Mitigating code-reuse attacks with control-flow locking. In: ACSAC'11 (2011)
5. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: ASIACCS'11 (2011)
6. Bosman, E., Bos, H.: Framing signals - a return to portable shellcode. In: SP'14 (2014)
7. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: CGO '03 (2003)
8. Carlini, N., Wagner, D.: ROP is Still Dangerous: Breaking Modern Defenses. In: SSYM'14 (2014)

9. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: CCS'10 (2010)
10. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: SSYM'05 (2005)
11. Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.R., Brunthaler, S., Franz, M.: Readactor: Practical code randomization resilient to memory disclosure. In: SP'15 (2015)
12. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In: SP'14 (2014)
13. Davi, L., Dmitrienko, R., Egele, M., Fischer, T., Holz, T., Hund, R., Nuernberger, S., Sadeghi, A.: MoCFI: A framework to mitigate control-flow attacks on smartphones. In: NDSS'12 (2012)
14. Davi, L., Sadeghi, A.R., Lehmann, D., Monrose, F.: Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: SSYM'14 (2014)
15. Drepper, U.: How to write shared libraries. `http://www.akkadia.org/drepper/dsohowto.pdf` (Dec 2010)
16. Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., Necula, G.C.: XFI: Software guards for system address spaces. In: OSDI'06 (2006)
17. Göktaş, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: Overcoming control-flow integrity. In: SP'14 (2014)
18. Hiroaki, E., Kunikazu, Y.: ProPolice: Improved stack-smashing attack detection. IPSJ SIG Notes pp. 181–188 (2001)
19. HTTP Archive: Http archive - interesting stats - average sizes of web sites and objects (2014), `http://httparchive.org/interesting.php?a=All&l=Mar%201%202014`
20. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A Safe Dialect of C. In: ATC'02 (2002)
21. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: SSYM'02 (2002)
22. Kuzentsov, V., Payer, M., Szekeres, L., Candea, G., Song, D., Sekar, R.: Code pointer integrity. In: OSDI (2014)
23. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: Automated Software Diversity. In: SP'14 (2014)
24. Le, L.: Exploiting nginx chunked overflow bug, the undisclosed attack vector (CVE-2013-2028) (2013)
25. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI'05 (2005)
26. MacManus, G., Hal, Saelo: Metasploit module nginx chunked size for CVE-2013-2028. `http://www.rapid7.com/db/modules/exploit/linux/http/nginx_chunked_size` (2013)
27. Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K.W., Franz, M.: Opaque Control-Flow Integrity. In: NDSS'15 (2015)
28. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In: PLDI'09 (2009)
29. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: CETS: Compiler Enforced Temporal Safety for C. In: ISMM'10 (2010)
30. Necula, G., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems (TOPLAS) 27(3), 477–526 (2005)

31. Nergal: The advanced return-into-lib(c) exploits. Phrack 11(58), http://phrack.com/issues.html?issue=67&id=8 (Nov 2007)
32. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI'07 (2007)
33. Niu, B., Tan, G.: Monitor integrity protection with space efficiency and separate compilation. In: CCS'13 (2013)
34. Niu, B., Tan, G.: Modular control-flow integrity. In: PLDI'14 (2014)
35. PaX-Team: PaX ASLR (Address Space Layout Randomization). http://pax.grsecurity.net/docs/aslr.txt (2003)
36. Payer, M., Gross, T.R.: Fine-grained user-space security through virtualization. In: VEE'11 (2011)
37. Payer, M., Hartmann, T., Gross, T.R.: Safe loading - a foundation for secure execution of untrusted programs. In: SP'12 (2012)
38. Philippaerts, P., Younan, Y., Muylle, S., Piessens, F., Lachmund, S., Walter, T.: Code pointer masking: Hardening applications against code injection attacks. In: DIMVA'11 (2011)
39. Pincus, J., Baker, B.: Beyond stack smashing: Recent advances in exploiting buffer overruns. IEEE Security and Privacy 2, 20–27 (2004)
40. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit Object-oriented Programming. In: SP'15 (2015)
41. SCO: System V Application Binary Interface, Intel386 Architecture Processor Supplement. http://www.sco.com/developers/devspecs/abi386-4.pdf (1996)
42. Seibert, J., Okhravi, H., Soederstroem, E.: Information leaks without memory disclosures: Remote side channel attacks on diversified code. In: CCS (2014)
43. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS'07 (2007)
44. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal War in Memory. In: SP'13 (2013)
45. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in gcc & llvm. In: SSYM'14 (2014)
46. van de Ven, A., Molnar, I.: Exec shield. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf (2004)
47. Wang, Z., Jiang, X.: Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: SP'10 (2010)
48. Xia, Y., Liu, Y., Chen, H., Zang, B.: CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In: DSN'12 (2012)
49. Zeng, B., Tan, G., Erlingsson, U.: Strato: A retargetable framework for low-level inlined-reference monitors. In: SSYM'13 (2013)
50. Zhang, C., Wei, T., Chen, Z., Duan, L., McCamant, S., Szekeres, L.: Protecting function pointers in binary. In: ASIACCS'13 (2013)
51. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: SP'13 (2013)
52. Zhang, M., Sekar, R.: Control Flow Integrity for COTS Binaries. In: SSYM'13 (2013)