Mitigating Code-Reuse Attacks with Control-Flow Locking

Tyler Bletsch Xuxian Jiang Vince Freeh
Department of Computer Science,
NC State University, Raleigh, NC, USA
{tkbletsc, xuxian_jiang, vwfreeh}@ncsu.edu

ABSTRACT

Code-reuse attacks are software exploits in which an attacker directs control flow through existing code with a malicious result. One such technique, return-oriented programming, is based on "gadgets" (short pre-existing sequences of code ending in a `ret` instruction) being executed in arbitrary order as a result of a stack corruption exploit. Many existing code-reuse defenses have relied upon a particular attribute of the attack in question (e.g., the frequency of `ret` instructions in a return-oriented attack), which leads to an incomplete protection, while a smaller number of efforts in protecting *all* exploitable control flow transfers suffer from limited deployability due to high performance overhead. In this paper, we present a novel cost-effective defense technique called *control flow locking*, which allows for effective enforcement of control flow integrity with a small performance overhead. Specifically, instead of immediately determining whether a control flow violation happens before the control flow transfer takes place, control flow locking lazily detects the violation after the transfer. To still restrict attackers' capability, our scheme guarantees that the deviation of the normal control flow graph will only occur *at most* once. Further, our scheme ensures that this deviation cannot be used to craft a malicious system call, which denies any potential gains an attacker might obtain from what is permitted in the threat model. We have developed a proof-of-concept prototype in Linux and our evaluation demonstrates desirable effectiveness and competitive performance overhead with existing techniques. In several benchmarks, our scheme is able to achieve significant gains.

1. INTRODUCTION

Computers are under constant threat of being compromised by increasingly sophisticated attackers. In the context of network daemons, attackers send maliciously crafted packets which exploit software bugs in order to gain unauthorized control. Research into preventing the existence of such bugs has been ongoing, but so far has failed to yield a silver bullet to the problem of exploitable security flaws. However, solutions targeting different aspects of the attack itself have had some success; this has led to an arms race between malicious attackers and security researchers.

One of the earliest attack techniques is the *code injection* attack, in which new machine code is written into the vulnerable program's memory, then a bug is exploited to redirect control flow to this new code. Fortunately, the protection technique known as W⊕X, which ensures that memory is either writable or executable (but not both), largely mitigates this attack [1]. However, attackers have responded by employing *code-reuse* attacks, in which a software flaw is exploited to weave control flow through existing code-base to a malicious end. For example, the *return-into-libc* (RILC) technique is a relatively simple code-reuse attack in which the stack is compromised and control is sent to the beginning of an existing libc function [2]. Often, this is used to call `system()` to launch a process or `mprotect()` to create a writable, executable memory region to bypass W⊕X.

To achieve greater expressiveness, a more sophisticated code-reuse attack known as *return-oriented programming* (ROP) was introduced [3]. In this technique, so-called *gadgets* (small snippets of code ending in `ret`) are weaved together in arbitrary ways to achieve Turing complete computation without code injection. In response to this threat, researchers developed defenses which relied upon particular attributes of this attack, such as the the frequency of `ret` instructions [4, 5] or reliance on the stack [6, 7, 8, 9, 10]. Unfortunately, recent evidence has revealed that code-reuse attacks need not rely on the stack or the `ret` instruction to govern control flow, negating these defenses [11, 12]. With that, it is not desirable to continue the trend of developing piecemeal defenses against each new code-reuse variant. Instead, there is a need to enforce *control flow integrity* (CFI), the property that a program's execution is restricted to the *control flow graph* (CFG) dictated by its source code. The CFI property was formalized by Abadi et al. in 2005; this work included a system capable of enforcing CFI in practice [13]. Unfortunately, this system has not seen significant production deployment possibly due to performance concerns.

In order to address this problem, we present an alternative technique, *control flow locking* (CFL), which achieves protection functionally equivalent to CFI enforcement with small performance overhead (e.g., negligible slowdown for many workloads). The key insight that allows for CFL's improved performance over earlier systems is that the validity check is performed *lazily*, i.e. after the actual control flow transfer has occurred. This leads to a better use of the

ACSAC '11 Dec. 5-9, 2011, Orlando, Florida USA

CPU's split L1 cache, as code need not be loaded as data (see Section 5.2 for details).

Control flow locking is analogous to mutex locking, except instead of synchronization, the lock is asserted to ensure correctness of the control flow of the application. Specifically, a small snippet of *lock* code is inserted before each indirect control flow transfer (`call` via register, `ret`, etc.). This code asserts the lock by simply changing a certain *lock value* in memory; if the lock was already asserted, a control flow violation is detected and the program is aborted. Otherwise, execution passes through the control flow transfer instruction onto the destination. Each valid destination for that control flow transfer contains the corresponding *unlock* code, which will de-assert the lock if and only if the current lock value is deemed "valid". If an invalid code is found (such as a lock from another point of origin), the process will be aborted. In our system, we further take additional precautions to ensure the lock value is not modified by other code, and that unintended instructions cannot be used (see Section 3). Given this, an attacker can now only subvert the natural control flow of the application at most once before being detected. Further, the instructions for system calls can have lock-verification code prepended to them so that this single control flow violation cannot be used to issue a malicious system call. This means that the only consequence of the attack are potential changes to the program's memory state, which is an ability the attacker is already assumed to have based on the threat model (see Section 3).

We have implemented a proof-of-concept control-flow locking prototype on a commodity x86 system. Though our current prototype mainly supports statically-linked programs, it has been shown to offer performance overhead competitive with existing techniques, achieving significant gains in several benchmarks. We believe control-flow locking represents a step further to address code-reuse attacks with a performance penalty small enough for possible deployment in real-world situations. The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 introduces the overall system design and Section 4 presents the implementation. Section 5 evaluates the system in terms of performance and correctness, and Section 6 discusses its implications as a whole. Section 7 concludes this paper.

2. BACKGROUND AND RELATED WORK

In this section, we first present a brief primer on the x86 architecture which readers familiar with that platform may safely skip. After that, we review recent code-reuse attacks as well as existing defenses.

2.1 Background on the x86

To understand the technique presented in this paper, it will be useful to review the technical details of the 32-bit x86 architecture, where our technique has been implemented.

First, note that the x86 assembly language presented is written in AT&T syntax. Registers are written with a percent sign (e.g. `%eax`), literals with a dollar sign (e.g. `$1`), and memory locations as plain symbols (e.g. `k`). Unlike Intel syntax, here the destination operands appear *last*, so `add %edx,%ecx` indicates $ecx \leftarrow ecx + edx$. Memory dereferencing is indicated either with an asterisk (e.g. `*%eax`) or with parenthesis (e.g. `(%eax)`).

Second, instructions are of variable length and are not aligned. This allows for the existence of *unintended code*: new interpretations of existing code achieved when jumping to a non-instruction boundary. For example, if an `add` instruction contains the bytes `0xff,0x10` in an operand, one could jump *into* that instruction at those bytes, and the CPU would now interpret the code as `call *%eax`.

Third, the x86 has a hardware-managed stack supported by two CPU registers: `esp`, the top-of-stack pointer, and `ebp`, the bottom-of-frame pointer. In addition to manual manipulation of stack memory, several x86 CPU instructions implicitly affect the stack. For our purposes, the most important are `call`, which pushes the next instruction's address onto the stack, and `ret`, which pops the stack and jumps to the popped address. These instructions are used to maintain control flow for function call and return, respectively. The `ret` instruction is referred to as an *indirect control flow transfer*, meaning that, based on the content of the stack, control flow can be redirected *anywhere*. This is what makes the `ret` instruction vulnerable to exploitation by the return-into-libc and return-oriented programming techniques—simply altering memory by means of a software exploit can lead to arbitrary control flow. In addition to `ret`, there are indirect variants of `call` and `jmp` instructions: they may take a register or memory reference as an operand, allowing a similarly unrestricted change of control flow.

Therefore, to guard against code-reuse attacks in the general case, it is necessary to not only protect `ret`, but also the indirect forms of `call` and `jmp`.

2.2 Code-Reuse Attacks

The return-into-libc (RILC) technique is one of the earliest forms of code-reuse attack. The simplest variant, which calls a single libc function, was documented as early as 1997 [2]. The technique was subsequently refined in order to allow for the chaining of multiple functions [14].

To achieve greater expressiveness from the attack, Shacham introduced return-oriented programming (ROP) in 2007 [3]. In this technique, the attacker seeks out small snippets of code within the existing codebase, each ending in a `ret`. These gadgets may perform memory load/store operations, arithmetic, logic, system calls, etc. By laying out malicious stack content full of ROP gadget addresses, the attacker can execute an arbitrary number of these gadgets. Further, because gadgets may affect the stack pointer itself, it is possible to construct conditional "branches", meaning that execution of return-oriented gadgets need not be linear. This allows for attacker-controlled Turing complete computation in the context of the vulnerable application. ROP has been demonstrated on a variety of platforms, from x86 [3] to RISC [15, 16, 17] to even embedded environments [18]. A variant of ROP has been adapted to the higher-level constructs in the PHP scripting language, culminating in a data stealing exploit as well as an attack on the PHP exception handler that is able to launch to a traditional machine-code-level code-reuse attack [19].

Return-oriented programming has also been applied to the kernel environment: a ROP-based kernel rootkit has been shown to negate all existing kernel integrity protections [20]. Also, this work demonstrated the feasibility of developing ROP attacks in an automated manner based on a ROP "compiler". Further, it has been shown that the codebase needed to achieve Turing completeness in a ROP attack need not be very large. Even the bootloader of an embedded device may have enough gadgets to implement a return-oriented rootkit, which has negative implications for software attestation techniques which seek to ensure program integrity [21].

In response to this, researchers examined the specific attributes that characterize a ROP attack in order to develop corresponding defenses. ROPDefender [7] rewrites binaries to maintain a shadow stack in order to verify each return address. This builds upon previous work in stack protection, including other shadow stack system [8, 9, 10], as well as canary-based stack protection such as StackGuard [22]. Systems like DROP [4] and DynIMA [5] can detect a ROP-based attack based on the short length of ROP gadgets, which results in a very high frequency of `ret` instructions being encountered. The return-less approach even goes so far as to systematically remove every `ret` instruction so that no gadgets can be found in the resulting binary [6].

In response to these defenses, code-reuse attacks were developed which did not rely on `ret`. Checkoway et al. introduced a technique of finding gadgets which end in sequences equivalent to `ret`, such as `pop X ; jmp X` or the Branch-Load-Exchange (BLX) instruction on ARM [12]. Bletsch et al. went further, establishing *jump-oriented programming* (JOP) as a means to maintain malicious control flow without any `ret`-equivalent instructions through the introduction of a special dispatcher gadget [11]. These attacks indicate the need for general defenses, because ad-hoc defense techniques that target specific attributes of specific attacks are insufficient to address the breadth of code-reuse attack techniques available. The existing work in this vein is discussed in the following subsection.

2.3 Existing Defenses

There are several defense techniques proposed to mitigate code reuse attacks. For example, Address-space layout randomization (ASLR) randomizes a process memory layout, making it difficult for an attacker to figure out what pointers to inject into the attack buffer [23, 24, 25, 26]. Unfortunately, ASLR shares its own limitations [14, 27]. In fact, a so-called *de-randomization attack* has been shown to defeat the popular PaX ASLR system in just 216 seconds [28]. Nevertheless, ASLR still raises the bar for attacks and has been widely adopted.

In order to guarantee protection in the face of code-reuse attacks, it is necessary to restrict indirect control flow transfers so that they can only arrive at certain destinations. One of the first techniques seeking to achieve this kind of protection was *program shepherding*, in which a security policy is applied to all control flow transfers [29]. It is implemented on top of a code interpreter framework with a dynamic optimization system to cache native translations of basic blocks. This approach achieves good protection, but had unacceptably high overhead for some workloads (up to 760% in one case).

Abadi et al. introduced the notion of Control Flow Integrity (CFI), which seeks to ensure that execution only passes through approved paths taken from the software's control flow graph [13]. To achieve this, at each indirect jump/call and return instruction, the target address is checked to see if it follows a valid path in the control flow graph. Unfortunately, this particular implementation of CFI suffered from large overhead, and has not seen wide deployment. However, the core idea of enforcing control flow integrity would effectively mitigate the threat of code-reuse attacks, provided it could be achieved at a reasonable cost. Subsequent work on the notion of CFI has allowed for additional security features, including Data Flow Integrity (DFI) [30] and others [31, 32, 33].

Recently, Onarlioglu et al. have introduced G-Free, another system aimed at addressing the threat of code-reuse in the general case [34]. This system works by systematically editing assembly code so that it does not contain indirect control flow transfers as unintended instructions. It can then secure the intended control flow transfers using data protection techniques similar to prior work (e.g. StackGuard [22], etc.). This technique appears to have much improved performance compared to CFI ($\sim 3\%$). However, it is difficult to discern how it performs with control flow intensive benchmarks, as the only evaluation of performance overhead for application-wide protection was on workloads in which control flow was not on the critical path (i.e. IO-bound or computation-kernel based workloads).

The control flow locking technique presented in this work shares many of the same goals as CFI and G-Free, in that it seeks to protect control flow from diversion by attackers. However, it does so in a very different way from these techniques, allowing greatly reduced overhead compared to CFI as well as G-Free in some cases.

3. DESIGN

假设

In this paper, we assume that the attacker can put a payload into memory and exploit a bug to overwrite some control data (a return address, function pointer, etc.), despite the presence of W⊕X protection. Further, we assume that the altered control data will be used at some point to govern control flow. Currently, this data can be used (or abused) to send control flow literally anywhere in the executable region, including unintended code, function entry points, or various return- or jump-oriented gadgets. This freedom on the part of the attacker is what we will be addressing.

The root problem of code-reuse attacks is their promiscuous use of control flow data in general (return addresses, function pointers, etc.). Therefore, there is a need to restrict this data to only valid targets as dictated by the program's control flow graph. To be specific, the control flow operations that must be protected are: (1) unintended code which happens to implement `ret`, `call`, or `jmp`; (2) `ret` instructions; and (3) indirect `call` and `jmp`.

To address the first category, we turn to existing software-fault isolation (SFI) technique introduced by McCamant et al. [35] and refined by the Google Native Client project [36]. This technique effectively eliminates the danger of unintended code. More specifically, because unintended code arises as a result of variable-sized non-aligned instructions, it can be eliminated by imposing alignment artificially. To achieve this, three changes can be applied to the software at the assembly code level. First, no instruction is permitted to cross an n-byte boundary. ? For the offending ones, no-ops will be inserted to re-align them. Second, all indirect control flow transfers are restricted to targeting n-byte boundaries. Third, all targets for indirect control flow transfers (e.g. post-call sites, function entry points, etc.) are forced to align to an n-byte boundary. These rules, when taken together, ensure that unintended code cannot be reached[1]. In practice, the value of n is a power of two, and control

[1] See [35] for a formal proof of this property.

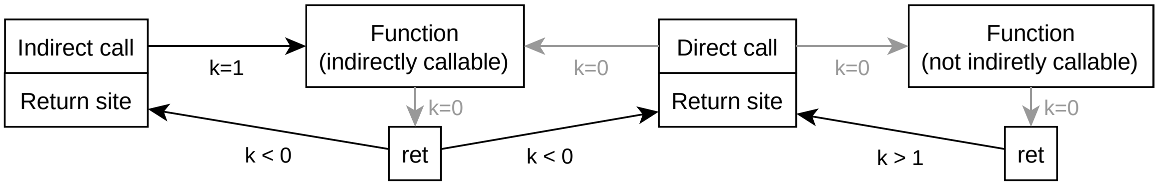


Figure 1: A simplified view of the control flow graph used in the CFL technique. Indirect `jmp` instructions, which generally arise from compiler optimizations, are not shown. The value of the control flow key `k` is shown next to each edge; grey edges do not require control flow locking and are simply present for completeness.

flow transfers are restricted by means of a simple bit mask. The value of n must be larger than any single instruction, and there is a trade-off between smaller n (more frequent instruction alignments) and larger n (longer no-ops before control flow transfers, increased pressure on the CPU instruction cache). In practice, $n = 32$ is a typical value that is chosen when the corresponding transformation is applied at the assembly phase of the build process.

With unintended code removed from consideration, we now focus exclusively on control flow transfers in the rest two categories, i.e., `ret`, indirect `call` and `jmp`. To this end, we apply a novel technique called *control flow locking*.

3.1 Control Flow Locking

To explain this technique, we begin with a degenerate variant of it called *single-bit control flow locking*. In this model, we first locate all indirect control flow transfer sites, which consists of `ret` instructions and the indirect variants of `jmp` and `call`. Before each of these sites, we insert "lock" code, which implements the following pseudo-code:

```
if (k != 0) abort();
k = 1;
```

Here, `k` is known as the *control flow key*, and is simply a word of memory at a fixed location. In this model, the value 0 means "unlocked" and 1 "locked". This is analogous to a mutex lock operation, though atomicity and waiting are not required for our purposes. At each valid indirect control flow target[2] found in the control flow graph, we apply the corresponding "unlock" operation:

```
k = 0;
```

In normal operation, every lock will immediately be followed by a transfer to a corresponding unlock. The attacker's ability to employ a code-reuse exploit, however, has been sharply limited. Control flow from an indirect transfer must pass through an unlock operation before encountering another indirect transfer. Because the only unlock operations correspond to valid transfer targets, this means using coarse-grained pieces of code, such as entire functions. Further, it will not be possible to apply RILC-style whole-function chaining as proposed in [14], because the so-called *esp-lifter* gadget used to connect the functions has been eliminated. Therefore, the only code eligible for use by the attacker are those which are valid indirect transfer targets and happen to themselves include a legitimate control flow transfer.

From the attacker's perspective, this selection is orders of magnitude smaller than the full gamut of gadgets normally available. Moreover, we can make a further addition to make it functionally equivalent to full enforcement of control flow integrity. The above locking algorithms merely ensure that the target of an indirect transfer must pass through *any* valid entry point before jumping again. However, we can extract additional information from the control flow graph to place finer granularity on our enforcement. To this end, `k` can be changed from a single bit to an integer, with values corresponding to paths along the control flow graph. We call this variant *multi-bit control flow locking*. In this model, the lock and unlock algorithms may be rewritten as:

```
lock(value):
    if (k != 0) abort();
    k = value;
unlock(value):
    if (k != value) abort();
    k = 0;
```

Here, `value` is a parameter determined at link-time based on the control flow graph, which limits control flow to valid paths specified in the program's source code. The manner in which the `k` value is selected is covered in the following section.

3.2 Restricted Control Flow Graph

The general meaning of "control flow graph" includes every basic block in the software as nodes and any labels or deviations from linear execution as edges. This includes conditional branches, direct jumps, etc. For the purposes of this work, however, we need only concern ourselves with indirect control flow transfers, i.e. those that jump to a location stored in memory or a register as opposed to in the instruction itself. Therefore, we use a restricted control flow graph which consists of the following classes of nodes: (1) entry points into functions; (2) locations in the code which may be the target for indirect `jmp` or `call`; (3) return sites; and (4) indirect `call` and `ret`.

Accordingly, the graph has the two types of edge: The first type represents the control flow resulting from a `ret` instruction or an indirect `call` or `jmp`. The second type is implied from a function's entry point to each of its `ret` instructions. In Figure 1, we illustrate the two types of edges in our control flow graph. For each black edge, the origin endpoint is the location of "lock" code, and the destination endpoint contains corresponding "unlock" code. The grey edges are direct transfers of control flow, and therefore do not require locking.

Control flow locking is a general mechanism to enforce restrictions on indirect control flow transfers – its accuracy depends on the manner in which `k` is selected at each node and the granularity with which it is matched at each destination. The degenerate *single-bit* variant discussed earlier

[2] The exact definition of an indirect jump target varies depending on the code in question. For statically linked code (including the OS kernel) and code destined to be a standalone executable, only those locations explicitly used as function pointers in the source code are considered targets. For dynamic library code, however, any exported function symbol is also a potential entry point.

`k` value	Meaning
`k` = 0	Unlocked.
`k` = 1	Indirect `jmp` or `call`.
`k` > 1	Return from a *non-indirectly-callable* function.
`k` < 0	Return from a *indirectly-callable* function.

Table 1: Possible `k` values in multi-bit CFL.

is equivalent to enforcing $k = 1$ for every black edge. In this work, we evaluate *multi-bit* CFL with a policy derived from static analysis of the source code's control flow graph. The possible values of `k` are enumerated in Table 1.

To protect `ret` instructions in a given function, the lock code before each `ret` sets `k` to a specific value based on which instructions may call it. The specific value is computed as follows. First, we determine the list of direct `call` instructions which refer the function in question. This list is hashed to a produce the value `d`. Second, we determine if the function may be called indirectly. In practice, this means finding if the function's symbol has been used in a data declaration or as an operand to a non-control-flow instruction, such as `mov`. This fact is stored in the boolean `indir`. Based on the above, assuming a word size of 32-bits, the value of `k` is computed as:

```
(indir ? 0x80000000 : 0) | (0x7FFFFFFF & d)
```

The lower 31 bits represent the caller hash `d` with the sign bit representing `indir`. This means that functions which may be called indirectly will have a negative `k` value, while those that cannot will have a positive `k`. (In the unlikely event that `d` = `indir` = 0, a positive `d` will be chosen.) Choosing the values of `k` in this way allows the comparison code to be written in the fewest x86 instructions possible (see Section 4).

In the implementation presented in Section 4, all indirect `call` and `jmp` operations share the `k` value of 1. This is due to a limitation in static code analysis: any symbol that represents a location in code which is also used as data may be stored and referenced in arbitrary ways, through multiple pointers, overlapping data structures, etc. As such, it is not possible to automatically identify which locations an indirect `call` or `jmp` may lead to in general. Therefore, we conservatively assume that any indirect control flow transfer may go to any code location whose symbol is used as data.

This is not a limitation of the CFL technique, however. If additional control flow information can be provided by the programmer or the higher level language (e.g., a more restrictive language than C), then CFL could readily make use of this information to enforce this new finer-grained control flow graph. As originally introduced by Abadi et al., assigning keys to indirect `jmp`/`call` control flow paths can lead to a problem of *destination equivalence* [13]. This occurs because two indirect call sites may have non-disjoint sets of potential destinations. For example, suppose X may call A or B, while Y may call B or C. In this case, list of callers of A, B, and C differ, but B must allow control to flow from either X or Y with a single `k` value. There are three possible solutions to this problem. First, when ambiguity is present, we may assign a single `k` value for all functions involved (e.g., X and Y would share a `k` value). Second, we may apply more fine-grained comparison, such as each destination checking a subset of bits of `k` (e.g., B only checks the lower 16 bits). Third, we may duplicate whole functions, providing different `k` values for the each (e.g., B is replicated as B'). These techniques can be combined depending on the precise structure of the indirect CFG. Due to limitations of static code analysis, however, the implementation presented treats indirect `jmp` and `call` instructions as equivalent. This granularity is sufficient to prevent the jump-oriented programming technique presented in [11], because it precludes the existence of jump-oriented gadgets.

The edges of Figure 1 have been annotated with values for `k`. At each edge origin, the lock code sets `k` to the specified value. At each target, the unlock code verifies that the value of `k` is one of those set by an incoming edge. For example, the return site after a direct call will verify that `k` is equal to the specific key value for the called function, whereas the return site after an indirect call will merely confirm that `k` < 0, as this confirms that control flow was transferred by a `ret` within an indirectly callable function.

Two additional considerations are needed to ensure that control flow locking cannot be bypassed. First, we must ensure that the value of `k` cannot be modified directly through code other than the lock and unlock routines. The x86 architecture has a feature that allows this to be achieved in a straightforward way – memory segmentation. Segmentation permits applications to have multiple separate memory maps which can be uniquely addressed via segment selector registers. Modern operating systems use a flat memory model, and therefore make little use of this feature, meaning that an entire segment register can be dedicated solely to deal with storing `k`. Because there is no unintended code and no explicit segment selection in application-generated code, the only code available to address `k` lies in the lock and unlock routines.

Second, we must ensure that the attacker cannot redirect control flow directly into a system call (e.g., a `sysenter` instruction). This can be achieved simply by prepending lock verification code to each system call instruction which will validate that `k` is in the unlocked state (0). This forces control flow for a system call to pass through the corresponding function entry point.

4. IMPLEMENTATION

To assess the performance impact of the CFL technique, an implementation was developed on a 32-bit x86 Debian Linux 5.0.4 system with an Intel Core 2 Duo E8400 3.0GHz CPU. Because the protection must be applied to complete software stack, a CFL-enabled libc was built based on dietlibc[3] version 0.32 [37]. In addition, gcc itself includes a static library called `libgcc` for inclusion in all binaries – this library contains helper functions unique to each hardware architecture. A CFL-enabled variant of this library was also produced. This implementation is based on statically linked binaries; Section 6 discusses how the technique can be extended to dynamically linked binaries.

4.1 Overview

The CFL system was implemented as two additional phases within the normal gcc build system: (1) an assembly language rewriter and (2) a small post-link patch-up phase. The

[3] This libc variant was designed to minimize code size, but the reason that it was selected for this work is its simplicity and adherence to best practices where assembly code is concerned (such as proper use of locally scoped symbols). These factors simply eased the implementation; there's no reason why a more common libc implementation, such as GNU libc, could not be used instead.

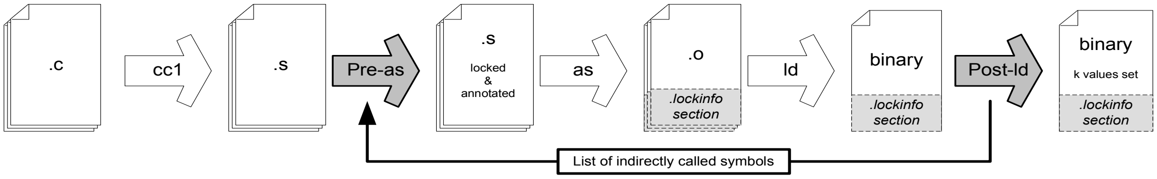


Figure 2: Modified gcc workflow. The grey *Pre-as* and *Post-ld* phases are the only additional steps in the build process.

complete workflow is diagrammed in Figure 2. The assembly language rewriter performs the vast majority of the work, encapsulating both the alignment transformations needed to eliminate unintended code as well as the core CFL transformations. Placing the transformation at the assembly phase allows both C and assembly-language code to be protected, but has the downside of requiring our system to reconstruct some semantic information, such as the call graph, control flow graph, and the list of symbols eligible to be called indirectly. Further, much of this information is not available at level of the individual assembly language files, so it was necessary to implement compilation as a two-pass process.

In the first pass, the assembly rewriter inserts lock and unlock code under the assumption that no symbol may be called indirectly. During this, records of each code symbol, symbol reference, lock operation, and unlock operation are noted in a new ELF section[4] called `.lockinfo`. Then, during the post-link phase, the `.lockinfo` can be used to determine all indirectly callable code symbols. This symbol list is exported for use in the second build pass.

During the second pass, the assembly rewriter can now use the list of indirectly callable code symbols to insert additional unlock operations as needed. As before, all lock and unlock operations are noted in the `.lockinfo` section. The `k` values used in these operations are simply dummy values, as the call graph is not yet known. During the post-link phase of the second pass, the call graph is available, and every `k` value can be computed. The system therefore uses the locations of the lock and unlock operations recorded in the `.lockinfo` to patch in the proper `k` values directly into the x86 code. At this time, the `.lockinfo` section can be discarded to reduce the executable size; it is not needed at runtime.

To summarize, the assembly rewriter phase will: (1) align instructions and function entry points on 32-byte boundaries and restrict control flow instructions to 32-byte boundaries; (2) note all symbol references and code labels in `.lockinfo`; (3) insert lock code before all indirect control flow transfers; and (4) insert unlock code before all indirect control flow destinations (including those found during the post-link phase of the first pass). And the steps undertaken by the post-link patch-up phase are: (1) use the `.lockinfo` to construct the call graph and identify all lock and unlock code locations; (2) make note of indirectly called symbols for use in the second pass; and (3) patch the binary with the `k` values computed for each function in all lock and unlock code.

4.2 Lock/unlock operations

The specific values for `k` (described in Section 3.2) were selected to allow the implementation of lock and unlock operations to be done efficiently. This is achieved by exploiting the difference between signed and unsigned comparisons on the x86. Figure 3 depicts the key assembly code transformations which insert lock and unlock code.

Figure 3(a) shows the lock code inserted before each `ret`. The first two lines ensure that `k` is 0, otherwise aborting with a lock violation error. The third line sets `k` to the proper value for this particular function. At assembly time, this `<key>` is simply set to a dummy value, which is later filled in during the post-link phase.

Figure 3(b) shows the corresponding unlock code for a direct call. This code will ensure that `k` is set to the proper value for the called function before clearing `k` back to 0.

Figure 3(c) shows two transformations. First, because this is an indirect call operation, a lock is inserted before the call. As before, this lock ensures that `k` is 0 (unlocked). It then sets `k` to the proper value for an indirect call (1). After the call returns, a special variant of the unlock code is inserted. This variant must verify that `k` contains a value corresponding to an indirectly callable function, i.e., a negative value. Therefore, the first two lines of this unlock will compare `k` to 0 and abort if $\texttt{k} \geq 0$. Otherwise, `k` will be unlocked.

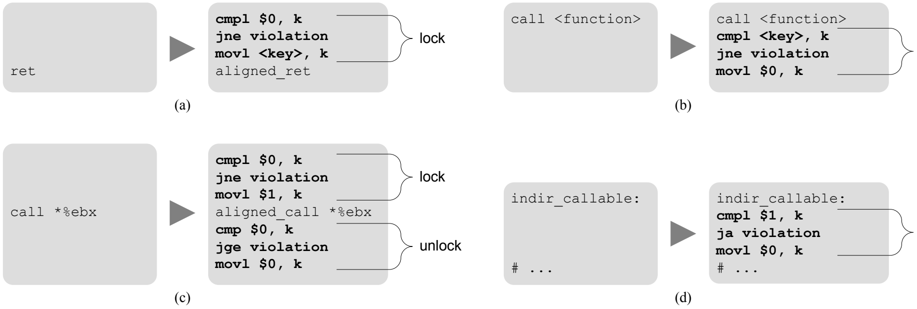
Figure 3(d) shows the code inserted at the site of a label which may be indirectly called or jumped to. This unlock code corresponds to the lock set in the first top half of Figure 3(c). This code may be reached via an indirect call or jump, in which case $\texttt{k} = 1$, or it may be accessed via a direct call or jump, in which case no locking has taken place and $\texttt{k} = 0$. Therefore, this unlock code must accept only those cases. To achieve this, we switch to using an unsigned compare. To determine if `k` is 0 or 1, we compare it to 1 and abort if and only if $\texttt{k} \overset{u}{>} 1$; otherwise, `k` is unlocked and execution continues.

4.3 Assembly language caveats

It is important to note that, in principle, the concept of a call graph only exists in the realm of higher level languages such as C. Assembly code need not respect this concept. For example, it is possible for one hand-coded (or compiler-optimized) function to jump or fall through into another without using a `call` instruction. In addition, the differentiation between a full-fledged function and a mere label in GNU assembly language is merely based on naming – local labels start with `.L`. However, hand-coders of assembly language need not follow this convention, as the only consequence for making all symbols global is some confusion when using the debugger.

Therefore, it was necessary to ensure that all manually written assembly code (such as that found in dietlibc) be

[4] ELF binary objects in Linux are composed of multiple sections, such a `.text` for code and `.data` for writable data. Arbitrary new sections can be introduced as needed and do not affect normal operation of the program. These sections are present in object files and linked together when building the final binary, at which time any symbols used are resolved.



(a)

```
ret
```

```
cmpl $0, k
jne violation
movl <key>, k
aligned_ret
```

(b)

```
call <function>
```

```
call <function>
cmpl <key>, k
jne violation
movl $0, k
```

(c)

```
call *%ebx
```

```
cmpl $0, k
jne violation
movl $1, k
aligned_call *%ebx
cmp $0, k
jge violation
movl $0, k
```

(d)

```
indir_callable:

# ...
```

```
indir_callable:
cmpl $1, k
ja violation
movl $0, k
# ...
```

Figure 3: Code transformations responsible for control flow locking. Some transformations for alignment are also shown: the macros `aligned_ret` and `aligned_call` are the same as the normal `ret` and `call` instructions, except they restrict the operand to align to a 32-byte boundary.

made to conform to the same rules as code generated by the C compiler (or at least have the exceptions noted explicitly). To this end, minor changes were made to dietlibc to explicitly indicate fall-through, and direct jumps between functions were detected and automatically annotated as well. When two functions have such a relationship, we say that they are *jump-connected*. Further, jump-connectedness is a transitive relationship, so if A and B are each jump-connected to C, then A is jump-connected to B, and vice versa. Detecting such cases is necessary, because a function may return *on behalf of* another. In this case, the `k` value for the return lock code must match for the two functions.

Therefore, where previously we have made reference to the list of callers of a function, a more accurate description would be the list of callers to all functions jump-connected to the one under consideration.

5. EVALUATION

In this section, we first present the security analysis on the protection offered by CFL. Next, we measure the performance overhead of our prototype.

5.1 Security Analysis

In order to analyze the security provided by CFL, we will review each possible destination for a `ret` instruction or indirect `jmp` or `call` that has been exploited. The possible targets under consideration are the nodes of our indirect CFG (`ret`, indirect `jmp`/`call`, function entry points, and return sites) as well as system call sites. Table 2 enumerates all possible destinations and their eventual outcomes.

In this table, the only outcomes that do not result in the software aborting are those on the valid CFG. The case of control flow arriving "before an indirectly callable function" requires clarification: because functions are n-byte aligned, the only code available *before* a function is the content of the previous function, and all control flow paths in that function must end in a `ret` (or an equivalent operation, such as a direct `jmp` to another function). Therefore, there is no lock-free code path available before a function entry point that would fall through into the function itself.

In Table 2, where "preceding code" may be executed, this cannot include system calls, so the only side effect is a change to program memory and CPU state. However, recall that the attacker's ability to alter CPU and memory state is already assumed, based on the threat model (presented at the start of this section). Therefore, an attacker attempting a code-reuse attack on a CFL-enabled binary achieves no greater control of the program than was provided by the original bug being exploited. In addition, we note that it is possible to mechanically determine if a binary has been properly compiled with CFL protection. This is a straightforward extension of the reliable disassembly method introduced in Native Client [36]. Once the alignment rules have been confirmed, the only extension needed is to verify that all indirect control flow transfers have corresponding lock and unlock code. Because 32-byte alignment eliminates the possibility of unintended code, this check is simply a straightforward scan of the disassembled binary. This means that system-wide CFL protection can be enforced by including such a verification routine in the OS binary loader.

5.2 Performance Analysis

To evaluate the performance impact of CFL, we built a number of C based benchmarks from the SPEC CPU 2000 and SPEC CPU 2006 suites, as well as some common UNIX utilities. The SPEC CPU benchmarks were run using their standard reference workloads. The workload for the UNIX utilities were similar to those used in the evaluation of G-Free [34]: `md5sum` computed the MD5 hash of a 2GB file, `grep` searched a 2GB text file for a short regular expression, and `dd` created a 4GB file on disk by copying blocks from `/dev/zero`. These applications were built using four different assembly rewriter algorithms:

- **None**: No changes made.
- **Just alignment**: Only the alignment rules needed to preclude unintended code are implemented.
- **Single-bit CFL**: The degenerate single-bit jump locking algorithm in which the only values for `k` are 0 and 1. The unlock code is simplified, as it need not check for a locked state, and will instead simple set `k` to 0.
- **Full CFL**: The complete control flow locking scheme.

Overhead was computed for the latter three algorithms compared to "None" as the base case; these results are presented in Figure 4. Overall performance impact can be divided into four categories.

First, many of the workloads (`mcf`, `milc`, `lbm`, `md5sum`, `grep`, and `dd`) exhibited negligible overhead. These applications likely did not perform a large amount of control flow compared to useful computation, i.e., control flow operations were not on the critical path. This may be due to their primary computation being implemented as coarse-grained, long-running functions, or (in the case of `dd`) another resource such as IO being on the critical path.

Second, the compression benchmarks `gzip` and `bzip2` incurred 2–3% overhead just due to alignment, then almost

If control flow is directed...	Then the system will...
at or before a `ret` instruction	run preceding code; abort due to pre-`ret` lock
at or before an indirect `jmp`/`call`	run preceding code; abort due to pre-`call` lock
before an indirectly callable function	abort due to a lock within the previous function
at a valid function entry point (for jmp/call)	proceed normally (valid control flow transfer)
at an invalid indirectly callable function	abort on unlock due to k mismatch
at a valid return site (for ret)	proceed normally (valid control flow transfer)
at an invalid return site	abort on unlock due to k mismatch
before a return site (i.e. before a direct call)	enter function; abort at the next lock operation
at or before a syscall	run preceding code; abort due to pre-syscall trap

Table 2: Possible paths of exploited control flow and their outcomes.

no additional overhead from the inclusion of CFL. That is, the no-ops and address masking operations involved in preventing unintended code accounted for almost the entirety of overhead for these workloads.

Third, the performance of `art` is an interesting case. In single-bit CFL, it incurs almost 5% overhead, but yields near-zero overhead for plain alignment and full CFL. This case is puzzling, and it may be related to an idiosyncrasy of working with the x86 assembly language and microarchitecture. Modern x86 CPUs are super-scalar out-of-order processors with complex branch and value prediction and multiple layers of cache. In addition, some instructions have multiple forms with differing lengths. For example, the conditional jump instruction can be short (encoded in 2 bytes for distances less than 128 bytes) or long (encoded in 6 bytes for larger distances). Therefore, it is possible for assembly-level modifications to have unexpected subtle effects on performance. For example, insertion of a three-instruction "lock" code may make a conditional jump go from short to long form, which in turn may ripple down and affect all subsequent alignment operations, which may in turn alter how the code fits into the CPU instruction cache or the contention for functional units within the ALU. It is very difficult to identify how these subtle changes may affect a given process's execution on a given CPU, so it is not clear that such effects are the necessarily culprit with `art`, but given that full CFL involves strictly *more* inserted code than single-bit CFL, yet has less overhead here, it is the best theory available to explain this case.

Fourth, some workloads (`gap`, `twolf`, and `sjeng`) exhibited significant overhead as more and more instructions were added to govern control flow operations. It is likely that these workloads make use of fine-grained control flow, such as calling many short-lived functions, in the course of their execution. This is supported through application profiling: the `gap` benchmark, which saw the largest CFL overhead, performed over 3.6×10^7 calls per second, whereas `mcf`, which had negligible overhead, performed only 6.5×10^5 calls per second. One interesting thing to note is that the CFL technique was applied with no modification to the C optimizer; it may be the case that adjustments could be made to the optimizer to reflect the newly increased cost of control flow operations to mitigate this overhead. For example, the logic that determines when a function should be inlined may need to be re-calibrated to reflect the increased cost of function calls. The question of how to mitigate the performance impact of CFL opens an interesting avenue for future work.

The CFL technique compares favorably against prior techniques. One of the highest overheads recorded for the control flow integrity (CFI) technique proposed by Abadi et al. [13], CPU2000's `gap` benchmark, saw 31% overhead. The CFL technique provides equivalent protection with 21% overhead for this workload. For the other benchmarks available for direct comparison (`bzip2`, `gzip`, and `twolf`), CFI achieved overheads between 0% and 5%; CFL achieves comparable results. A direct comparison between these figures isn't strictly possible, as the CFI work was conducted on a different OS and CPU, and the source code for the system is not available. However, CFL likely compares favorably in the general case because it involves a similar amount of instrumentation, but incurs strictly less L1 data cache pressure compared to CFI. The reason for this is as follows. The x86 L1 cache is split between instructions and data. In CFI, when the destination is checked, memory at the jump target must be loaded as data in the cache. Then, after the comparison succeeds and control moves to the target, the same memory must be loaded again, this time into the instruction side of the L1 cache. In contrast, the CFL technique places the key values as immediate operands within the instructions being executed, removing the need for this double-load behavior. In short, CFL executes similar operations at each control flow transfer while removing needless data cache pressure.

With regard to the G-Free system, making a comparison is difficult because the published evaluation in that work was limited to IO-bound and computation-kernel workloads[5] [34]. As such, without a direct comparison of G-Free across a larger number of workloads, it is difficult to speak generally about the relative performance it versus CFL. Nevertheless, for the metrics that are available, the CFL technique is competitive. Four benchmarks are shared between this work and G-Free's evaluation: `gzip`, `grep`, `dd`, and `md5sum`. For `gzip`, CFL achieved roughly equal overhead to G-Free: 3%. For the others, CFL achieved essentially zero overhead (within the standard deviation), whereas G-Free incurred between 0.6% and 2.6% overhead[6]. These results are promising, but inconclusive. It is possible that neither technique is superior in all cases; there may be a trade-off that makes one of the two schemes more efficient for a given application.

[5] Specifically, the evaluation did end-to-end protection for six workloads: `gzip`, `grep`, `dd`, `md5sum`, `ssh-keygen`, and `lame`. In addition, a wider selection of benchmarks was tested with their technique protecting libc *only*, not the application code. Because control flow is seldom on the critical path within libc, these measurements fail to characterize the performance of the G-Free system with a workload rich in control flow operations, so it is not possible to conclusively compare those workloads to systems like CFL or CFI.

[6] No information on test repetitions or measurement variance was provided in [34].

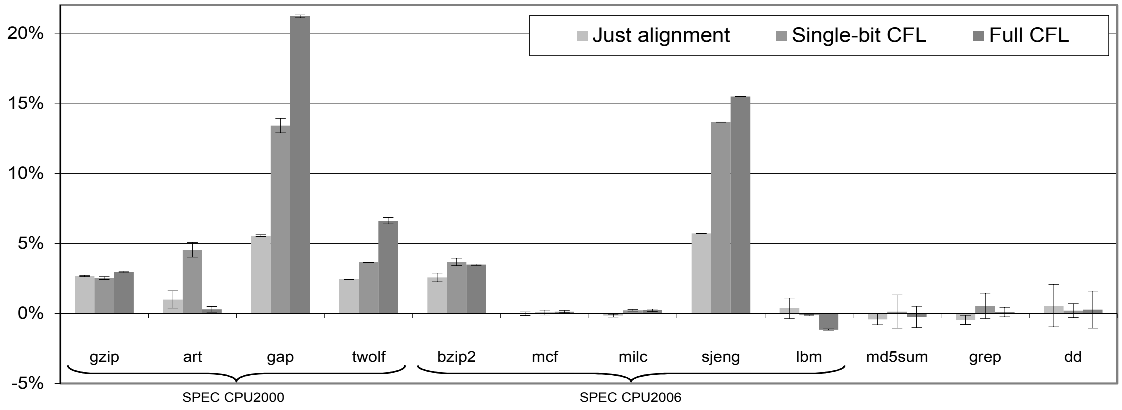


Figure 4: Performance overhead of various forms of the CFL technique.

6. DISCUSSION

It is important to clarify the precise security benefit that the CFL scheme offers: it constrains the path of execution to the software's control flow graph, allowing at most one violation, and guaranteeing that this violation cannot be used to construct a malicious system call directly. In the following, we further elaborate our system and examine possible limitations in our prototype.

First, we highlight that the protection is only as good as the control flow graph being enforced. In this implementation, an automatically derived graph was used. This graph imposed tight restrictions on direct `call`s and their corresponding `ret`s, and limited indirect `call` and `jmp` instructions to those entry points which were used indirectly in the assembly code. The indirect `call`/`jmp` protection could be improved if the programmer or higher-level language provided more precise insight into how indirect control flow transfers were to be used.

Second, our threat model presumes that the exploit in play can be used to alter the application's memory. The CFL technique is only intended to mitigate the risk of a code-reuse attack, which exploits the program's control data. However, as Chen et al. correctly observed, "Non-control-data attacks are realistic threats" [38]. That is, malicious attacks on plain variables can yield significant security problems, including unauthorized access and privilege escalation. Further, one can easily imagine a scenario in which a pure-data attack could even lead to malicious Turing complete behavior without altering control flow. Interpreters are a straightforward example of this risk: overwriting the "code" being interpreted yields arbitrary attacker-controlled behavior. As such, the CFL technique is not a *silver bullet* for all the dangers posed by software exploits; it mitigates the threat posed specifically by code-reuse attacks.

The implementation presented in Section 4 was based on statically linked binaries. This was primarily for ease of implementation; there is no reason in principle why the technique could not be applied to dynamically linked binaries. From a conceptual perspective, the conversion is straightforward: (1) the analysis of the call graph currently performed in the post-ld phase would be moved to the runtime linker, and (2) because we do not know at build time which exported symbols may be called indirectly, unlock code would be inserted in all exported functions' entry points, to be removed as needed by the runtime linker. Of course, the call graph of different programs would lead to different `k` values within dynamic shared libraries, which is problematic, as the library code pages would no longer be able to be shared. One possible solution would be to add a layer of indirection to the lookup of `k` values: instead of embedding the value directly in the instruction, a lookup into a per-process table could be substituted. It is not immediately clear what the additional overhead of this modification would be and we leave it to our future work.

The G-Free technique, which modifies assembly code to remove unintended indirect control flow transfers, may present an interesting extension to CFL. Currently, the problem of unintended code is solved by imposing alignment based on prior work on sandboxing [35, 36]. However, it may be possible to replace this technique with the branch-removal algorithm employed by G-Free, while leaving the actual control flow lock/unlock code as-is. This hybrid technique may have performance benefits which could make it more attractive than the current G-Free or CFL systems. Examining this possibility is another research question for us to explore.

7. CONCLUSION

This paper presented a novel defense against code-reuse attacks called *control flow locking* (CFL). This technique ensures that the control flow graph of an application is deviated from no more than once, and that this deviation cannot be used to craft a malicious system call. CFL works by performing a "lock" operation before each indirect control flow transfer, with a corresponding "unlock" operation present at valid destinations only. The technique has been implemented in practice on a commodity x86 system, and it has been shown to offer performance overhead competitive with existing techniques, achieving significant gains in several benchmarks. CFL represents a step further to mitigate code-reuse attacks with a small performance penalty.

8. REFERENCES

[1] Wikipedia. W^X. http://en.wikipedia.org/wiki/W^X.

[2] Solar Designer. Getting around non-executable stack (and fix). *Bugtraq*, 1997.

[3] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *14th ACM CCS*, 2007.

[4] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Drop: Detecting return-oriented programming malicious code. In *5th ACM ICISS*, 2009.

[5] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-oriented Programming Attacks. In *4th ACM STC*, 2009.

[6] Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace, and Sina Bahram. Defeating return-oriented rootkits with return-less kernels. In *5th ACM SIGOPS EuroSys Conference*, April 2010.

[7] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. Technical Report HGI-TR-2010-001, Horst Görtz Institute for IT Security, March 2010.

[8] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *21st IEEE ICDCS*, April 2001.

[9] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *10th USENIX Security Symposium*, 2001.

[10] Vendicator. Stack Shield: A "Stack Smashing" Technique Protection Tool for Linux. *http://www.angelfire.com/sk/stackshield/info.html*.

[11] Tyler Bletsch, Xuxian Jiang, Vince Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *6th AsiaCCS*, March 2011.

[12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming Without Returns. In *17th ACM CCS*, October 2010.

[13] Martín Abadi, Mihai Budiu, Úlfar Erilingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *12th ACM CCS*, October 2005.

[14] Nergal. The Advanced Return-into-lib(c) Exploits: PaX Case Study. *Phrack Magazine, Volume 11, Issue 0x58, File 4 of 14*, December 2001.

[15] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *15th ACM CCS*, pages 27–38, New York, NY, USA, 2008. ACM.

[16] Felix "FX" Lidner. Developments in Cisco IOS Forensics. In *CONference 2.0*, November 2009.

[17] Tim Kornau. *Return oriented programming for the ARM architecture*. Master's thesis, Ruhr-Universität Bochum, January 2010.

[18] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. In *EVT/WOTE 2009, USENIX*, August 2009.

[19] Stefan Esser. Utilizing Code Reuse/ROP in PHP Application Exploits. In *BlackHat USA*, 2010.

[20] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *19th USENIX Security Symposium*, August 2009.

[21] Daniele Perito Claude Castelluccia, Aurélien Francillon and Claudio Soriente. On the Difficulty of Software-Based Attestation of Embedded Devices. In *16th ACM CCS*, New York, NY, USA, 2009. ACM.

[22] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security*, page 5, 1998.

[23] PaX Team. PaX ASLR Documentation. http://pax.grsecurity.net/docs/aslr.txt.

[24] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. *12th USENIX Security*, 2003.

[25] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. *14th USENIX Security*, 2005.

[26] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. *22nd SRDS*, October 2003.

[27] Tyler Durden. Bypassing PaX ASLR Protection. *Phrack Magazine, Volume 11, Issue 0x59, File 9 of 18*, June 2002.

[28] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address Space Randomization. *11th ACM CCS*, 2004.

[29] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *11th USENIX Security Symposium*, August 2002.

[30] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *7th USENIX OSDI*, November 2006.

[31] Úlfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *7th USENIX OSDI*, 2006.

[32] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *28th IEEE Symposium on Security and Privacy*, May 2008.

[33] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-Granularity Software Fault Isolation. In *22nd ACM SOSP*, October 2009.

[34] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *ACSAC*, 2010.

[35] Stephen McCamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. In *MIT Tech Report MIT-CSAIL-TR-2005-030*, 2005.

[36] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[37] Felix von Leitner et al. dietlibc. http://www.fefe.de/dietlibc/.

[38] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security*, pages 177–192, 2005.