CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks

Xi Chen
Vrije Universiteit Amsterdam
x.chen@vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

***Abstract*—Code diversification is an effective strategy to prevent modern code-reuse exploits. Unfortunately, diversification techniques are inherently vulnerable to information disclosure. Recent diversification-aware ROP exploits have demonstrated that code disclosure attacks are a realistic threat, with an attacker able to *read* or *execute* arbitrary code memory and gather enough gadgets to bypass state-of-the-art code diversification defenses.**

In this paper, we present *CodeArmor*, a binary-level system to harden code diversification against all the existing read-based and execution-based code disclosure attacks. To counter such attacks, *CodeArmor* virtualizes the code space to completely decouple code pointer values from the concrete location of their targets in the memory address space. Using a combination of *run-time randomization* and pervasively deployed *honey gadgets*, code space virtualization probabilistically ensures that only code references that can legitimately be issued by the program are effectively translated to the concrete code space. This strategy significantly reduces the attack surface, limiting the attacker to only code pointer gadgets that can be leaked from data memory. In addition, unlike existing leakage-resistant code diversification techniques that provide similar security guarantees, *CodeArmor* requires no access to source code, hypervisors, or special hardware support.

Our experimental results show that *CodeArmor* significantly raises the bar against existing and future attacks, at the cost of relatively low average performance overhead (6.9% on SPEC and 14.5% on popular server programs, and even lower—roughly halving such average overheads—when operating aggressive inlining optimizations at the binary level).

1. Introduction

Today's code-reuse attacks (CRAs) are critically dependent on discovering the memory that contains useful snippets of code—or *gadgets*. Armed with such knowledge, attackers divert a program's control flow to such snippets and chain them together to construct their payloads. For instance, memory disclosures that leak code pointers enable attackers to bypass defense mechanisms like ASLR and exploit binaries using a variety of code-reuse attacks such as ROP [79] and JOP [23]. Even in the presence of (fine-grained) *code diversification* [31], a single code pointer suffices if an attacker is able to probe the memory contents of the target process dynamically using, for example, a JIT-ROP strategy [82], [34].

In detail, traditional memory disclosures are based on explicit reads, where the contents of code pages/code pointers are leaked directly from memory. More recent attacks read memory pages indirectly, using time-based side channels [78], [57], [39]. However, attackers may also probe the target program by blindly executing code at some address until they encounter behavior that corresponds to a known gadget [15]. Whatever the method used, memory disclosure is a *conditio sine qua non* for modern code-reuse attacks. Phrased simply, all code-reuse attacks need to know where to transfer control to stitch together a payload. In the *ideal* case where no memory disclosures are possible, code diversification offers *perfect* protection against code reuse—potentially stronger than active defenses such as CFI [7], which have been recently targeted by a variety of attacks [21], [49], [35], [22].

In this paper, we present *CodeArmor*, a new binary-level solution which counters diversification-aware code-reuse attacks by *virtualizing* a program's diversified code space to prevent any code pointer stored in memory from revealing the location of the corresponding concrete code space, and continuously *randomizing* the mapping to the concrete code space to ensure strong information hiding guarantees against brute-force attacks (as opposed to the weak guarantees provided by plain ASLR [39]). The intuition is that code executes at constantly changing addresses, and all code pointers in memory require linear translation prior to use in program-issued control transfers and do not even point to concrete code pages until then. Thus, an attacker leaking data pages will only disclose (nontranslated) code pointers that do not allow the attacker to (directly or indirectly) find additional gadgets from the corresponding code pages.

With *CodeArmor* deployed, *read-based* attacks like JIT-ROP [82], [34] are no longer effective, since code pages themselves can never leak by design. Even if attackers leak a nontranslated code pointer from data pages, *execution-based* attacks can only run the corresponding code if the program first translates the pointer. The only way to do so is by means of the program's legitimate control transfers (and not those blindly derived from misaligned instructions), already slowing down brute-force attacks. Moreover, attackers cannot easily modify the code pointer and probe the rest of the code space using a BROP-like attack, because the code itself is

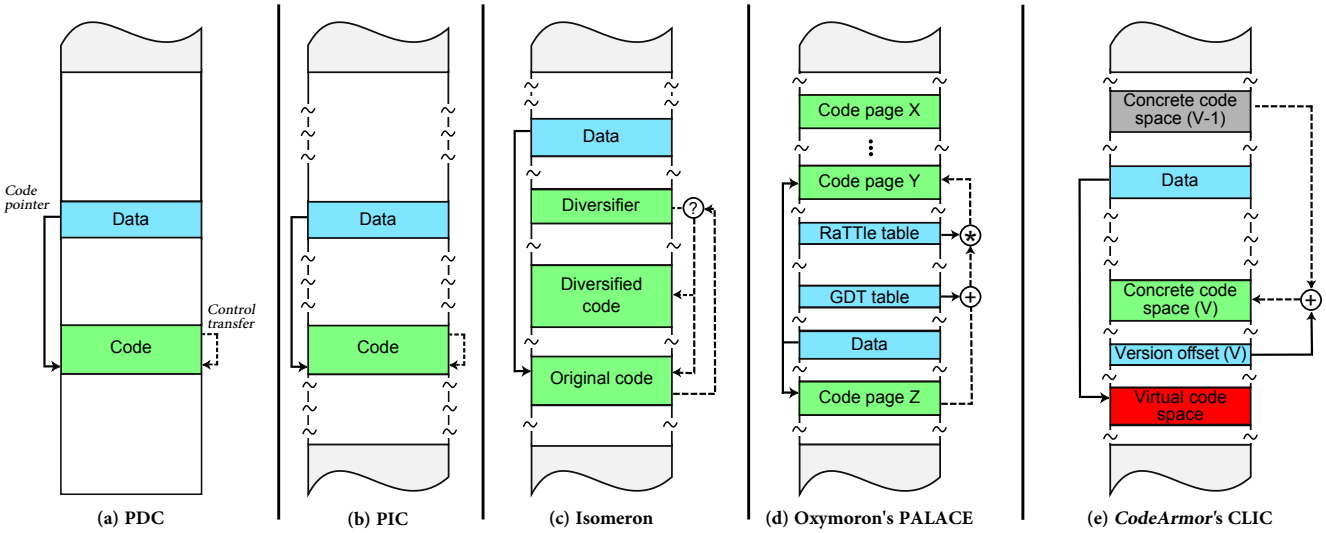


Figure 1. Comparison of different code space organizations.

highly diversified and many of the potential targets for control transfers in reality point to pervasive *honey gadgets* that raise an alert when targeted. *CodeArmor*'s design efficiently ensures that an attack will on average hit such honey gadgets a number of times before disclosing the first real gadget, providing a strong probabilistic defense against arbitrary execution-based attacks.

In other words, *CodeArmor*'s design probabilistically ensures that only "live" code pointers that can be leaked from data memory can be used (as-is) as individual gadgets by attackers, significantly reducing the attack surface. Unlike existing leakage-resistant code diversification techniques that provide similar security guarantees [43], [42], [27], [30], [83], [91], [64], [10], [9], [18], *CodeArmor* works entirely at the binary level without any need for source, special hardware support, or modifications to the underlying software stack (i.e., OS or hypervisor).

While our techniques are more general, *CodeArmor* specifically focuses on protecting C binaries for the x86_64 platform. Such binaries are representative of widely deployed security-sensitive server programs, which have repeatedly proven vulnerable to modern and sophisticated disclosure attacks [15], [78], [39]. In addition, *CodeArmor* can be used to enhance state-of-the-art binary-level CFI techniques [85] designed to counter function reuse attacks [77] (which only rely on data disclosure). This is to reduce traditional CFI's target sets to only the code pointers that can be effectively leaked from data memory (rather than all the dynamically computed code pointers indiscriminately, similar to source-level per-input CFI [68]).

Contributions. We make the following contributions:

- We present a new code space organization to support *virtualized* code pointers and periodically *re-randomize* the corresponding concrete code space mappings. We demonstrate that such techniques are ideally suited to countering code disclosure attacks.
- We demonstrate the effectiveness of our techniques in *CodeArmor*, a new defense solution against diversification-aware code-reuse attacks. *CodeArmor* operates entirely at the binary level and requires no cooperation from the underlying hardware and software stack.
- We evaluate *CodeArmor* on both standard benchmarks and popular server programs, showing that it provides a strong probabilistic defense—ensuring that even sophisticated *execution-based* code disclosure attacks will hit an average of at least 3 honey gadgets before disclosing the first real gadget—with low average performance overhead—6.9% on SPEC and 14.5% on popular server programs, and only 3.2% and 8.2% (respectively) when operating aggressive inlining optimizations at the binary level. Our performance is comparable to or faster than other binary-level security solutions [80], [38], [95], [94] and significantly faster than traditional run-time randomization systems at low (μs) re-randomization latencies.

2. Threat Model

We assume a strong threat model where an attacker can interact with the target program repeatedly, exploiting vulnerabilities that allow arbitrary reads, writes, and control-flow diversions. We also assume the most permissive class of programs that keep restarting forked worker processes after a crash (e.g., real-world server programs). To understand how *CodeArmor* stops attacks, we first review state-of-the-art code disclosure techniques available to our advanced attacker.

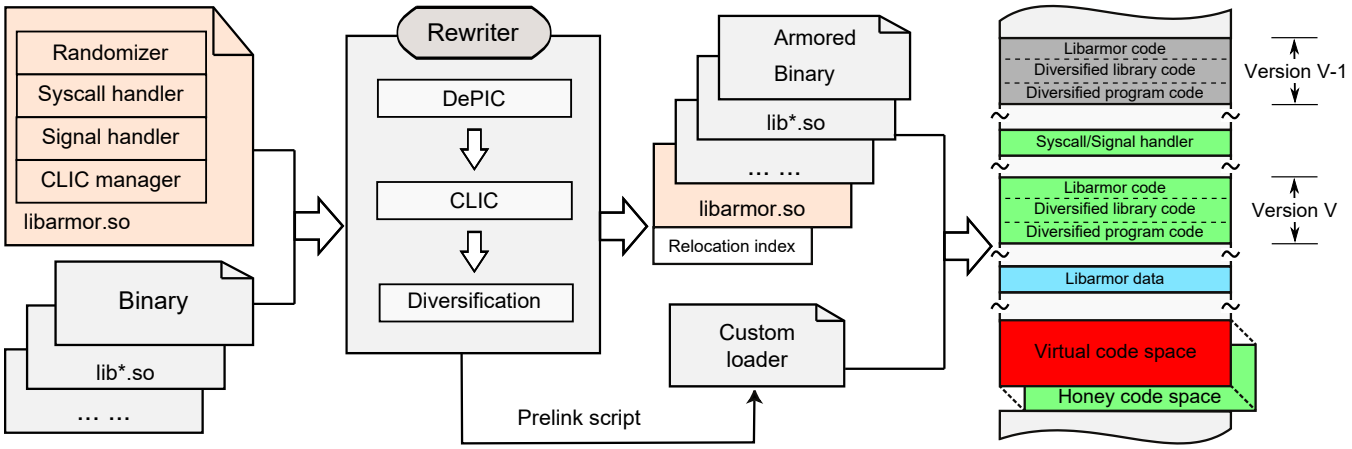


Figure 2. *CodeArmor*'s high-level overview

2.1. Read-based disclosure attacks

These attacks start by leaking a code pointer from the data space (*known-pointer* attack) or guessing a valid code space address (*blind* attack). Either way, the attacker locates a code page in memory and reads its content using an arbitrary read vulnerability. Recent attacks have exploited either direct memory disclosure [82], [34] or indirect reads using timing attacks [78]. By reading the code page, the attacker obtains all gadgets in it, but also control-transfer targets to locate even more code pages. Upon finding enough gadgets, the attacker can link them together and launch a traditional code-reuse attack. Key requirements for such a diversification-aware code-reuse attack are that gadgets found in the code space must be *readable* and remain *stable* throughout the attack.

2.2. Execution-based disclosure attacks

These attacks blindly execute code at a memory address and observe the execution behavior to disclose diversified gadgets in the code space [15], [78]. Attackers rely on a control-flow diversion vulnerability to redirect execution to a given code pointer (leaked or guessed in a *known-pointer* or *blind* attack, respectively), measure observable side effects to disclose the executed gadget (e.g., crashes), and repeatedly expand the search to surrounding addresses until enough gadgets are found. Key requirements for such a diversification-aware code-reuse attack are that gadgets found in the code space must be *stable* across failures and yield *nonanomalous* side effects (lest an IDS or a sysadmin detects the attack).

3. Concrete Layout Independent Code

One of the key ingredients to the protection offered by *CodeArmor* is its organization of the code space. Figure 1 compares different code space organizations, detailing their properties in relation with code disclosure attacks. Position-Dependent Code (PDC) and Position-Independent Code (PIC) are the standard code organizations used in real-world programs. In PDC-based binaries (Figure 1a), the code space is loaded at a fixed address. This enables an attacker to indiscriminately launch read-based and execution-based known-pointer disclosure attacks. In PIC-based binaries (Figure 1b), the code space can be loaded at any address in memory. This forces the attacker to leak pointers from data memory before reliably launching any known-pointer attacks, but it does not prevent disclosure attacks from succeeding in general.

Isomeron [34] (Figure 1c) and Oxymoron [10] (Figure 1d) are two recent software-only research solutions that propose new code organizations to mitigate disclosure attacks. Isomeron clones the original code space into a diversified code space, allowing a run-time diversifier to randomly switch the execution between the two spaces at every control transfer. Since the control flow randomly selects targets from the two code spaces, gadgets leaked from one (or the other) code space can no longer be reliably chained together to launch CRAs. With a code space switching probability of 0.5 at every control transfer, however, a repeated code-reuse attack based on, say, 5 chained gadgets—typically sufficient to mount a generic ret2libc attack on x86_64 [15]—requires only 32 attempts on average. Oxymoron's PALACE organization, in contrast, segments the code space into a number of randomly allocated code pages and forces control transfers to go through a RaTTle table that stores the real location of the targets. This strategy only eliminates control-transfer target information from code pages, limiting the expansion step of read-based disclosure attacks. This is, however, insufficient to prevent an attacker leaking multiple code pointers from gathering a sufficient number of gadgets from the corresponding code pages [34].

CodeArmor's CLIC (*Concrete Layout Independent Code*) in Figure 1e, finally, splits the original (diversified) code space into a virtual and concrete code space, completely decoupling code pointers stored in data memory from the concrete location of their targets in the address space and translating the former into the latter *only* at each valid control transfer. In addition, to ensure the concrete code

space remains "*hidden*" and its contained concrete gadgets *unstable*, the translation layer continuously switches to a new concrete code space version allocated at a random memory location. This design prevents attackers from effectively exploiting data memory leaks to start off code disclosure attacks and hinders all the illegal code references functional to mount read-based and execution-based disclosure attacks. This leaves only individual live code pointer-based gadgets directly disclosed via data leaks to attackers.

4. Overview

Figure 2 presents *CodeArmor*'s high-level overview. Binaries (and all their shared libraries) are instrumented by *CodeArmor*'s *Rewriter* and *Libarmor* run-time library and the address space reorganized according to a new layout known as CLIC (*Concrete Layout Independent Code*). A CLIC organization splits the original (diversified) code space into a virtual and concrete code space and completely decouples code pointers in data memory from the concrete location of their targets in the address space, requiring a translation of the former into the latter only at each valid control transfer. In other words, even the disclosure of code pointers in memory will not reveal the actual location of the target code. Moreover, to ensure the concrete code space remains "*hidden*" and its contained concrete gadgets *unstable*, the translation layer continuously switches to a new concrete code space version allocated at a random location.

The rewriter, in turn, internally relies on four sequential binary instrumentation passes: *DePIC*, *CLIC* and *Diversification*. The *DePIC* pass preprocesses the code (and data) to replace references using PC-relative addressing with corresponding references using absolute addressing. This is necessary to virtualize code pointers and ensure the concrete code space can be safely relocated at runtime.

The *CLIC* pass instruments all the legitimate control transfer instructions (except relative ones) to ensure their targets are linearly translated from the virtual to the concrete code space right before jumping to the destination during the execution[1]. In addition, the *CLIC* pass ensures *Libarmor* can transparently interpose on system calls (*Syscall handler*) and signal handling (*Signal handler*) at runtime to "hide" the CLIC organization to the underlying operating system. Finally, the *Diversification* pass performs fine-grained code diversification to ensure the virtual/concrete code layout remains unpredictable to the attackers even in the case of known-pointer attacks.

To run an instrumented program binary, users need to load the binary in *CodeArmor*'s *Custom loader*. The loader first relies on the pregenerated *Prelink script* to relocate code from the binary and all the libraries in a memory-contiguous *virtual code space*. This step is essential to enforce a CLIC memory layout and crucial for *CodeArmor*'s run-time performance. Next, the loader yields control to *Libarmor*, which allows the *CLIC manager* to finalize the

1. This software-based translation can be best described as a level of indirection on top of the MMU's virtual-to-physical translation.

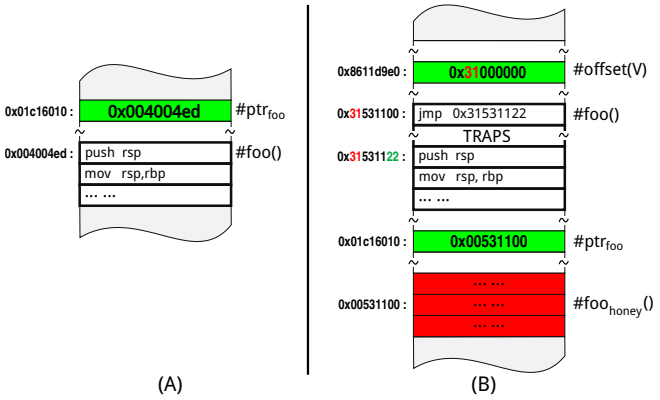


Figure 3. Runtime memory layout: baseline vs. *CodeArmor*

CLIC memory layout and start the program execution. The manager first relies on the *Relocation index* to dynamically patch all the absolute addresses introduced by the *DePIC* pass and fully retain *PIC*'s random relocation semantics. Once the virtual code space has been set up, the manager creates a (randomly allocated) clone to initialize the first *concrete code space* version at an initial offset from the virtual code space. Next, the manager intersperses the original code pages (still mapped at the virtual code space address) with pervasive *honey gadgets*—dummy gadgets that disclose no code layout information when *read* and raise alerts when *executed* by an attacker—effectively creating an identity mapping of the entire virtual code space into a *honey code space*.

Figure 3 demonstrates a runtime memory layout comparison between normal execution and execution protected by *CodeArmor*. During normal execution (Figure 3A), a function pointer ptr_{foo} directly stores the concrete address of the function *foo()*. Once such address is leaked, the attacker can further read the content of function *foo()* and mount a JIT-ROP attack. However, when the execution is protected by *CodeArmor* (Figure 3B), such attacks are no longer possible. First, the function pointer ptr_{foo} no longer holds the concrete address of function *foo()*. Instead, it contains a virtual code space address pointing to a clone of a diversified version of function *foo()* ($foo_{honey}()$), which is full of honey gadgets. Attempts to read or execute such gadgets will raise alerts caught by *CodeArmor*. In legitimate execution paths, in contrast, *CLIC* instruments all indirect control-flow transfer instructions by adding an offset (*offset(V)*) to their virtual code space target and locating the corresponding concrete code space address. In addition, to prevent an attacker from reusing any callsite or function entry gadgets, the diversification pass inserts dedicated random-sized gaps. These gaps are filled with *TRAPS* which redirect the control flow to the honey code space.

Since the *CLIC* instrumentation normally translates all the *legitimate* program-issued control transfers from the virtual to the concrete code space, we do not encounter the (honey) identity mappings except for attacker-initiated *misaligned* control transfers (i.e., originating in misaligned instructions and thus not instrumented to perform virtual-to-concrete

translations)). In particular, an attacker attempting a *read-based known-pointer* attack and using a pointer to the virtual code space to seed wild data reads will ultimately read useless data (instead of concrete code) from the honey code space. Alternatively, an attacker attempting an *execution-based known-pointer* attack and using a pointer to the virtual code space to seed wild code execution will ultimately cause misaligned control transfers, redirect execution to the honey code space, and trigger alerts. Furthermore, since all the code pointers stored in memory point to the virtual code space, no known-pointer attack can disclose the location of the concrete code space by design.

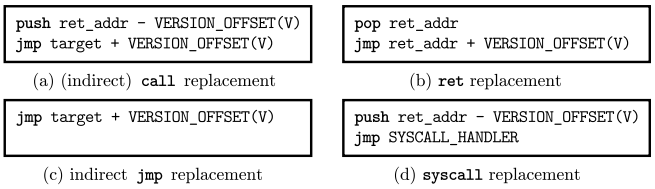
To further "hide" the concrete code space location to blind attacks, the CLIC manager deploys the *Randomizer*, a background thread which continuously creates new (randomly allocated) concrete code space versions and instructs the *CLIC* instrumentation to efficiently switch the execution accordingly. Once run-time randomization is enabled, the manager can finally yield control to the program and start off the execution.

5. Rewriter

5.1. DePIC pass

The goal of the *DePIC* pass is to eliminate all the PC-relative references from program and library code. This is crucial since PC-relative references are detrimental to the CLIC organization. PC-relative data references require code memory at a fixed offset from data memory. This would, however, allow arbitrary data pointer leaks to immediately reveal the location of the concrete code space. In addition, this would prevent *CodeArmor* from relocating the concrete code space during the execution, hindering run-time randomization. Similarly, PC-relative code references require an identity mapping between code memory and computed code pointer values. This would, however, allow arbitrary code pointer leaks to immediately reveal the location of the corresponding concrete code target. In addition, this would prevent code pointers from pointing into anything other than the concrete code space, hindering the fundamental assumption behind CLIC. By replacing all the PC-relative references with absolute ones, the DePIC pass can instead guarantee that (i) both code and data references are concrete-code-location-agnostic and (ii) code references stably point to the virtual code space.

To locate all the PC-relative references, the DePIC pass scans all the instructions in program and library code and checks the addressing mode in the `ModR/M` byte. On x86 memory-accessing instructions, the `ModR/M` byte specifies the operands and the addressing mode, while the `SIB` byte indicates the scale, index, and base register. Without even having to fully decode the individual instructions, the DePIC pass simply modifies the existing `ModR/M` and `SIB` bytes to specify absolute addressing and replaces the original displacement with a corresponding absolute address. The individual absolute addresses are computed relatively



(a) (indirect) `call` replacement (b) `ret` replacement

(c) indirect `jmp` replacement (d) `syscall` replacement

Figure 4. CLIC instrumentation's overview

to a reference base address and their location stored in a *Relocation index* embedded in the binary. This allows the CLIC manager to quickly locate and patch all the precomputed absolute references at load time (i.e., for the binary and dynamically linked libraries) and runtime (i.e., for dynamically loaded binaries/libraries, using a *just-in-time* rewriting strategy). This step is necessary to load the virtual code space at a random address on a per-program basis—similar to regular *PIC* code. Note that the DePIC pass operates exclusively on *PIC* code—used in the libraries and increasingly common in program code on x86_64 platforms. *PDC* code—already using absolute addressing—if present, is simply left untouched, effectively resulting in a split virtual code space organization with a nonrelocatable *PDC*-derived region and a fully relocatable *PIC*-derived region at runtime.

5.2. CLIC pass

The DePIC pass guarantees that all the code pointers stored in data memory contain absolute addresses that ultimately refer to the virtual code space at runtime. Since the concrete code space is identical to the virtual code space (an offset away), the goal of the *CLIC* pass is simply to ensure that *all* such virtual code pointers are linearly translated to a target in the concrete code space *only* when referenced in legitimate control transfers. Figure 4 illustrates the instrumentation operated by the *CLIC* pass for this purpose. In the figure, `VERSION_OFFSET(V)` refers to the global offset—maintained in *Libarmor data* memory and hidden to the attacker—between the virtual code space and the currently executing concrete code space version `V`.

The `call` instrumentation (Figure 4a) first translates the concrete return address into its virtual counterpart (necessary for all the code pointers stored in memory) and pushes the resulting address onto the stack. Next, it translates the virtual target address into its concrete counterpart (necessary for all the control transfer targets) and jumps to the resulting address. This second translation step is only actually required for indirect calls. Direct calls and jumps simply transfer control to a concrete target relative to the current (concrete) instruction pointer and thus require no virtual-to-concrete translation.

The `ret` instrumentation (Figure 4b) pops the virtual return address off the stack, translates it into its concrete counterpart, and jumps to the resulting address. The indirect `jmp` instrumentation (Figure 4c) is similar, but the target is explicit in the code. The `syscall` instrumentation (Figure 4d), finally, pushes the (translated) post-syscall return address and jumps to a dedicated *Syscall handler*. The *Syscall handler*,

along with a *Signal handler*, are the only code snippets allocated by *CodeArmor* in a separate (special) concrete code space, unaffected by run-time randomization and with no virtual code space mapping. This is necessary to support concrete kernel-to-user control transfers without requiring kernel modifications.

The `syscall` instrumentation also enables *CodeArmor* to interpose on selected syscalls. In detail, *CodeArmor* interposes on the `clone` syscall to allocate/initialize *Libarmor data*, the *Syscall handler*, and the *Signal handler* in a random location of the address space on per-process basis, start off the *Randomizer* on per-process basis, and initialize *CodeArmor*'s thread-local state on per-thread basis. In addition, *CodeArmor* interposes on the `signal` and `sigaction` syscalls to ensure all the signal handler invocations are proxied through the *Signal handler*. Finally, *CodeArmor* interposes on `execve` to perform *just-in-time* rewriting (and caching) of external binaries not known statically. A similar strategy is adopted by the *Custom loader* when loading external libraries at `dlopen` time.

Finally, the CLIC pass instructs the rewriter to generate a *Prelink script* at the very end of the binary instrumentation process. *CodeArmor*'s *Prelink script* specifies the load address for each instrumented library in the program. This facilitates the task of the run-time components to generate a memory-contiguous virtual and concrete (PIC) code space, significantly increasing the efficiency of run-time randomization—only one (PIC) region to remap at each randomization cycle—and, as by product, of the program itself—improved code locality and the ability to leverage huge code pages to reduce iTLB misses.

5.3. Diversification pass

The goal of the *diversification pass* is to implement effective code diversification techniques to make the location of the individual gadgets in the virtual code space (and, isomorphically, in the concrete code space) unpredictable. We note that, by preventing concrete code pointers from being actively stored in data memory, CLIC alone already provides some protection against read-based known-pointer attacks—a leaked virtual code pointer does not directly disclose information on the location of the concrete code space. Leaked virtual code pointers, however, can still indirectly disclose the location of other gadgets in the virtual code space, which the attacker can chain together by piggybacking on legitimate control transfers.

To counter indirect disclosure attacks, the diversification pass implements a cost-effective diversification strategy, which ensures randomly sized gaps are added before and after each possible code pointer stored in data memory. While arbitrary diversification passes can, in principle, be supported in *CodeArmor*, more sophisticated diversification strategies do not necessarily yield much higher entropy and can also thwart compiler optimizations hurting run-time performance [56].

Three classes of (virtual) code pointers can be stored in data memory: function pointers, return addresses, and (indirect) jump targets. To counter *known-function-pointer* attacks, the diversification pass permutes the functions in the binary, adds randomly sized gaps between functions and at individual function entry points, similar to prior source-level fine-grained ASLR strategies [31]. To counter *known-return-address* attacks, the diversification pass adds randomly sized gaps before and after each possible callsite. *known-jump-target* attacks, finally, can be handled similarly by adding randomly sized gaps before and after each possible indirect jump target. Finally, to mitigate the impact of CRAs that rely only on data disclosure, *CodeArmor* also reduces the number of live code pointers by hiding well-known read-only code pointer tables (i.e., jump tables and the GOT) inside the code space, similar to [30].

Randomly sized gaps are designed to satisfy a number of requirements. First, they should not directly affect existing code or compiler optimizations. This is ensured by starting each gap with a relative `jmp` causing existing code to simply jump over the gap during the execution. Second, they should not generate new gadgets that can be used by the attacker. This is simply ensured by using a `nop` sled to size the gap. Third, they should actively discourage the attacker from launching execution-based brute-force attacks to disclose the size of the gap. This is ensured by ending each gap with a misaligned (and thus nontranslated) `ret` control transfer instruction. This design guarantees that each brute-force attempt that lands on the gap will redirect execution to the honey code space and immediately raise alerts. Fourth, they must guarantee that all the pointers to the original code still work correctly after shifting the instructions. For PIC pointers (already identified and virtualized by the *dePIC* pass), this simply requires updating their value to reflect layout changes in the virtual code space. PDC pointers, however, cannot be accurately identified and updated in the same way—due to the imprecision of conservative pointer scanning, which can result in false positives even on 64-bit architectures [54]. To address this concern, the *diversification pass* conservatively scans (and overapproximates) the set of code pointers, extends gaps in the diversified code as necessary to ensure their values always target instructions inside gaps, and replaces the targeted instructions with jumps to the shifted target.

To control the gap randomization entropy, *CodeArmor* can be configured with a predetermined maximum gap size ($G_{\max}$). To select the default $G_{\max}$, the diversification pass adopts another cost-effective strategy, which simply guarantees that a gap-disclosing brute-force attack driven by a known pointer raises, on average, *more* alerts than a generic blind attack—effectively discouraging known-pointer execution-based attacks. While arbitrary $G_{\max}$ values could be supported, increasing the gap randomization entropy can also negatively affect code locality and memory usage [31].

To select a suitable $G_{\max}$ for our purposes, the diversification pass needs to first estimate the expected number of alerts triggered by a blind attack. Although multiple valid gadgets are generally necessary to launch a code-reuse attack on x86_64, we conservatively consider only the alerts triggered before the attacker can find the first potential legal gadget. This is equivalent to computing the expected number of misaligned (and thus nontranslated) control transfers

encountered by a blind attack before executing a legitimate (aligned and thus translated) one. Hence, assuming an attacker attempting to divert control flow using randomly selected virtual code pointers (execution-based blind attack assumption), the number of alerts before finding a first potential legal gadgets H_{blind} follows a stop-at-first-failure *negative hypergeometric distribution*, with $N = T_{\text{leg}} + T_{\text{mis}}$ elements (using *leg* and *mis* to indicate *legitimate* and *misaligned* control transfers, respectively) and T_{mis} successes. This translates to the following expected number of alerts $\overline{H}_{\text{blind}}$:

$$\overline{H}_{blind} = \frac{T_{\text{mis}}}{N - T_{\text{mis}} + 1} = \frac{T_{\text{mis}}}{T_{\text{leg}} + 1} \quad (1)$$

Thus, after scanning the binary for legitimate (T_{leg}) and misaligned (T_{mis}) control transfers, the diversification pass can simply compute $\overline{H}_{\text{blind}}$. To discourage known-pointer gap-disclosing attacks, the diversification pass conservatively selects the default $G_{\max}$ such that $\overline{H}_{\text{known}} = 2\overline{H}_{\text{blind}}$. Similar to H_{blind}, H_{known} follows a stop-at-first-failure *negative hypergeometric distribution*, but with $N = G_{\max}$ elements and $G_{\max}$–1 successes. This, again, translates to the following expected number of alerts $\overline{H}_{\text{known}}$:

$$\overline{H}_{\text{known}} = \frac{G_{\max} - 1}{N - (G_{\max} - 1) + 1} = \frac{G_{\max} - 1}{2} \quad (2)$$

Thus, the diversification pass can simply compute the desired default maximum gap size $G_{\max}$ as follows:

$$\overline{G}_{\max} = 2\overline{H}_{\text{known}} + 1 = 4\overline{H}_{\text{blind}} + 1 = \frac{4T_{\text{mis}}}{T_{\text{leg}} + 1} + 1 \quad (3)$$

6. Runtime System

6.1. Custom loader

CodeArmor's custom loader is a modified version of the default loader which sets up a consistent execution state and initializes the other run-time components. The loader needs to first allocate the *Syscall handler*, the *Signal handler*, and *Libarmor data* in a random location of the memory address space—similar to *CodeArmor*'s `clone` instrumentation at process-creation time. *Libarmor data* is initialized with data that remain hidden to the attacker, including the current `VERSION_OFFSET(V)`, the address of the *Syscall handler*, the address of the *Signal handler*, and *CodeArmor*'s own TLS. To guarantee that the location of *Libarmor data* itself remains hidden, the loader (and later *CodeArmor*'s thread-creation time instrumentation) pins one register for *Libarmor data*'s base address. On Linux x86_64, the register of choice is `%gs`, which is available for extensions using the `arch_prctl` syscall. After the inital setup, the loader—with the help of the *Prelink script*—loads the binary and libraries into memory, and yields control to the CLIC manager to initialize the code space.

6.2. CLIC manager

The goal of the CLIC manager is to translate the previously loaded code space into a proper CLIC organization and start a CLIC-enabled program execution. As a first step, the manager relocates the (former) PIC portion of the code space to a random location of the memory address space and relies on the *Relocation index* the identify and patch all the absolute addresses introduced by the *DePIC* pass accordingly. At the end of the process, the reserved code region (or regions, if a PDC portion is present) of the address space marks the *virtual code space*. To create the first version of the *concrete code space*, the manager simply clones the reserved code region into a new region mapped at a random location and updates the `VERSION_OFFSET(V)` accordingly.

To ensure that physical memory assigned to the concrete code space is reused across versions—important to reduce memory usage and eliminate unnecessary code cache misses during run-time randomization—the manager creates a private shared memory segment for the concrete code space and stores its key in *Libarmor data*. The same segment can be concurrently mapped into the memory address space at multiple locations referring to the same underlying physical pages. To implement its run-time randomization strategy, for instance, the *Randomizer* needs to maintain a double mapping for two consecutive concrete code space versions.

To create the *honey code space*, finally, the manager overwrites the reserved code region—the identity mapping for the virtual code space, reachable only by nontranslated control transfers—with *honey gadgets*. Our *honey gadgets* are pervasively deployed in the honey code space (and nontranslated control trasfers reaching them scattered throughout the concrete code space), in contrast to prior software booby-trapping efforts that argued for a performance-security tradeoff [29]. Honey gadgets are designed to satisfy a number of requirements. First, they should never disclose information whenever read-based attacks attempt to read from virtual code pointers (automatically landing on the honey code space). Second, they should always raise alerts whenever execution-based attacks attempt to execute a nontranslated control transfer (automatically landing on the honey code space). Third, they should not interfere with legitimate program code. Finally, they should not cause significant memory usage increase.

To satisfy the first two requirements, the manager simply selects illegal instructions as honey gadgets and pervasively deploys them over the honey code space. *CodeArmor*'s *Signal handler*, which already proxies all the application-specified signal handlers is instructed to special-case the `SIGILL` signal (i.e., illegal instruction) to determine whether the originating instruction was part of the honey code space. If so, the handler immediately raises an alert. If the originating instruction did not match a honey gadget, in turn, the signal is simply forwarded to the application-specified signal handler (if any). Note that this is necessary to preserve application transparency, given that it is not uncommon for real-world applications (e.g., Apache httpd) to register a `SIGILL` handler. When an alert is raised, a policy decides when to notify an external IDS/sysadmin and another policy decides when to

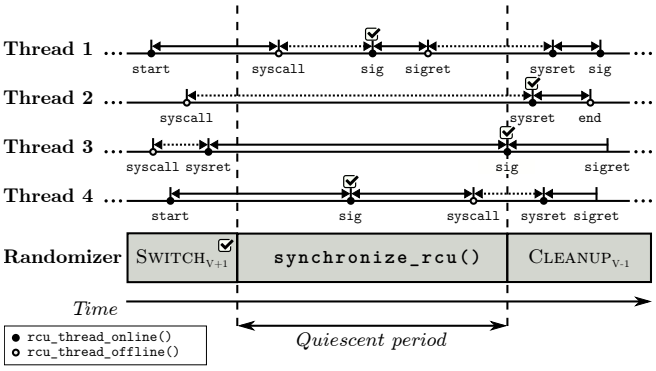


Figure 5. A sample run-time randomization run

shut down the application entirely. By default, *CodeArmor* sends notifications after 1 alert and shuts down the entire application after $\overline{H}_{\text{blind}}$ (the expected minimum number of alerts before finding a potential gadget).

To satisfy the third requirement, the honey code space needs to preserve the integrity of legitimate program-issued reads from mixed code/data memory (a common idiom generated by modern compilers). For this purpose, *CodeArmor* relies on its underlying binary analysis framework to identify mixed code/data memory and mark data-dedicated areas in the code space as *immutable*. Immutable areas (only) are prevented from being overwritten by honey gadgets, allowing program-issued reads to these areas to succeed and be served from the honey code space. Since an attacker can possibly fingerprint these areas by reading from honey code space, the Diversification pass ensures they are also surrounded by random gaps.

To satisfy the fourth requirement finally, the illegal instructions that implement honey gadgets are simply encoded using `\0xE` bytes, which guarantee the vast majority of the honey code space to be filled with `\0xE` pages (*honey pages*)—except for those few mixed code/data pages. To avoid unnecessarily wasting physical memory, the manager actually implements such honey pages by simply overmapping existing memory mappings with inaccessible (i.e., `PROT_NONE`) pages and instructs *CodeArmor*'s *Signal handler* to special-case the `SIGSEGV` (segfault) signal as done for the `SIGILL` signal above.

After setting up the honey code space, the manager initializes the *Randomizer* in a background thread and allows the program to start off the execution.

6.3. Randomizer

The goal of the *Randomizer* is to continuously re-randomize the location of the concrete code space, effectively "hiding" its location to the attacker. Even a blind read-based attack that happens to guess the location of the current concrete code space version correctly, should be quickly exposed to a new re-randomized concrete code space version when either attempting to launch a code-reuse attack or, more realistically, expand its search probing for multiple gadgets.

CodeArmor's run-time randomization strategy is designed to satisfy three key requirements. First, it should safely and transparently support arbitrary multithreaded program binaries. Second, it should minimally perturb program execution, with marginal performance/memory overhead and scalability impact. Third, it should provide low-latency re-randomization cycles, resulting in a frequently re-randomized concrete code space. To satisfy the first requirement, a naive strategy would simply maintain one local copy of the concrete code space for each thread and allow threads to periodically and independently re-randomize their own code space. Unfortunately, this strategy is detrimental to our second requirement, with per-thread concrete code space versions increasingly polluting the virtual memory address space (and degrading TLB performance) for higher thread counts (poor scalability) and also yielding a larger number of gadgets at the attacker's disposal (poor entropy).

To overcome problems with the naive strategy, the *Randomizer* opts for a different design, with a shared concrete code space version `V`, a global offset `VERSION_OFFSET(V)`, and a background re-randomization thread redirecting the execution from one version to the next. To satisfy the first requirement (thread-safety), however, concurrent accesses to `VERSION_OFFSET(V)` (and the corresponding concrete code space version) have to be properly guarded to ensure program/background thread synchronization. To address this challenge, our randomization strategy protects accesses to `VERSION_OFFSET(V)` using (user-level) Read-Copy-Update (RCU) [37]. RCU provides a scalable synchronization mechanism between a single *writer* (i.e., the background thread) and multiple *readers* (i.e., program threads), guaranteeing nearly-zero read-size performance overhead and low-latency write-side updates. These characteristics satisfy all our requirements.

For our purposes, we selected the *QSBR* flavour of `liburcu` (linked against `libarmor.so`), which implements the most efficient known user-level RCU strategy. This strategy requires each reader to periodically announce a *quiescent state* (e.g., using `rcu_thread_offline` and `rcu_thread_online` primitives), notifying the writer of the end of their *read-side section*. The writer, in turn, can wait for a *quiescent period* (i.e., using the `synchronize_rcu` primitive), that is blocking until all the active read-side sections terminate.

This mechanism can be efficiently used by the background thread to update the `VERSION_OFFSET(V)` to a new concrete code space version, wait for all the program threads to transition to the new version, and clean up the old one. Program threads, in turn, can be instrumented to announce a quiescent state when they `start` or receive a `sig` signal and to announce an extended quiescence state when they `end` or a `syscall` is in progress. System calls, in particular, are ideal extended quiescent points, given that the program threads can, in principle, suspend their execution in the kernel for an arbitrarily long period of time. This strategy is exemplified in Figure 5, with all the program threads switching to the new version by the end of the quiescent period with the exception of *Thread 2*. The latter, in an extended quiescent state induced

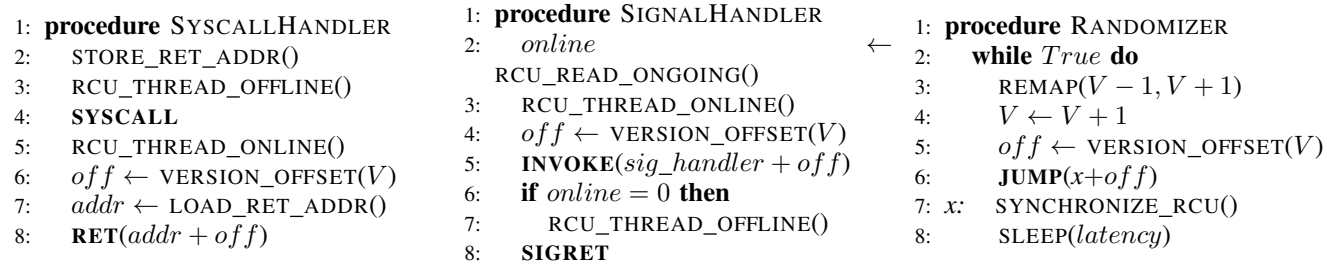


Figure 6. *CodeArmor*'s run-time randomization protocol

by a `syscall`, however, automatically switches to the new version when execution later returns to user mode (`sysret`).

Figure 6 presents an overview of our run-time randomization protocol, with the instrumentation required in *CodeArmor*'s *Syscall handler*, *Signal handler*, and *Randomizer*. The *Randomizer*'s background thread runs in an endless loop, each iteration implementing one randomization cycle and introducing extra delay to control the *latency*—currently defaulting to 0, given that we have observed no performance benefits for larger values.

Each loop iteration starts off with remapping the previous concrete code version `V-1`—which the background thread defers cleaning up to the next cycle—into the next version `V+1` at a random address space location. Since the concrete code space is memory-continuous (only split if PDC code is present), remapping can efficiently be implemented with a few system calls in the worst case—reducing latency and mode-switching costs. Next, the background thread redirects all the program threads to the new version (updating `VERSION_OFFSET(V)`) and immediately switches to the new concrete code space itself. By now, some program threads may be already running the new version, others may be still on the old one (both still mapped in memory). To ensure the old version can be safely cleaned up at the next iteration, the background thread relies on `synchronize_rcu` to wait until all the threads have switched to the new concrete code space.

The *Syscall handler*, in turn, saves the virtual return address previously pushed by the `syscall` instrumentation in *CodeArmor*'s own TLS, enters an extended quiescent state (`rcu_thread_offline`) and issues the actual syscall trap to enter kernel mode. When returning to user mode, it terminates its extended quiescent state (`rcu_thread_online`), acquires a new version offset, and jumps to the current concrete return address counterpart. The *Signal handler*, finally, saves the quiescent status of the current thread, enters a new read-side section, and invokes the application-specified signal handler using its virtual reference. Before invoking the application-specified signal handler, the instrumentation temporarily switches to a RCU-free copy of the *Syscall handler*, ensuring the quiescent status of the current thread is unchanged if the application happens to issue a short-lived system call in its own signal handler. When done, the *Signal handler* restores the original quiescent status of the current thread and `sigreturns`.

RCU semantics guarantees that the current version (and the underlying concrete code space) is stable as long as a program thread is *online*. During *offline* execution periods, however, the current version can be concurrently switched by the background thread at any instant. The implementation addresses this concern by ensuring that *only offline* periods of the *Syscall handler* (lines 3-5, and its RCU-free clone) and of the *Signal handler* (lines 2-4,7—the implementation ensures `sigreturn` is actually called during an online period by deferring the `rcu_thread_offline` call at line 7) are not affected by the *Randomizer*. This design leaves exactly 4 concrete landing pads stable across randomization cycles. To ensure these landing pads do not yield a usable gadget set, we remove all the unintended gadgets by using alignment instructions similar to [71]. This leaves only 2 intended gadgets: a `syscall` and a `jump *r11` gadget. Both gadgets are relevant, but effectively unusable without the attacker discovering other gadgets to control the data flow. When probing for other gadgets in unrelated locations, in turn, *CodeArmor*'s design ensures the target program will crash with extremely high probability. Even with forking applications that automatically recover from crashes, the CLIC manager will automatically allocate these gadgets and the small (typically one page) *Libarmor data* in entirely new random locations when setting up the new process context.

7. Implementation

We implemented *CodeArmor* on the Linux (x86_64) platform. The static rewriter is implemented as a Dyninst [11] (v8.2.1) extension in 2,403 lines of code. The run-time components (*Libarmor* and the *Custom loader*) are implemented in C (and some inline assembly) in 815 lines of code. Our current prototype can support generic 64-bit ELF binaries, with three main limitations. The first limitation is the inability to support C++ exceptions, directly inherited by Dyninst. The second limitation is the inability to support dynamically generated and self-modifying code. This is not an inherent limitation, as Dyninst's run-time writer can, in principle, be used to address these concerns, but not without explicit knowledge of the program under analysis. Both limitations are not fundamental and can be addressed with more engineering effort. A more fundamental limitation is the inability to support PIC code sharing across processes, which has a generally negative memory usage impact in system-wide deployment scenarios [10]. If memory usage is of concern, a simple solution to this problem is to eliminate

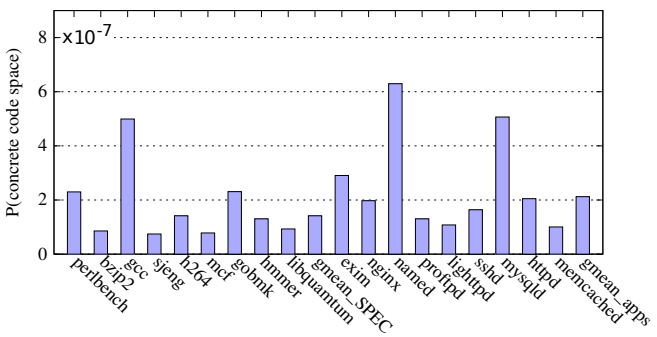


Figure 7. P(*concrete code space*) in a blind attack

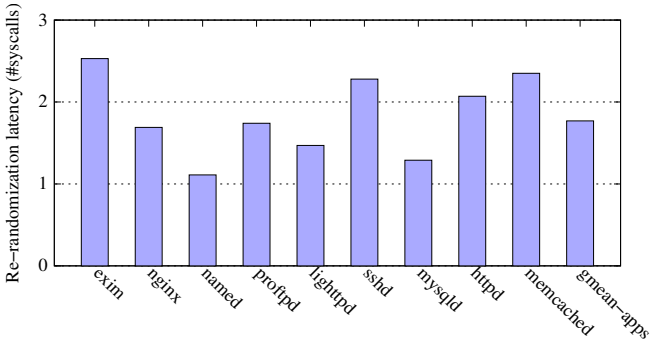


Figure 8. Average server re-randomization latency (#syscalls)

the *Relocation Index*, but this would also reduce all the execution-based attacks to known-pointer attacks.

8. Evaluation

We evaluated *CodeArmor* on an Intel i7-3632QM clocked at 2.20 GHz with 256 KB per-core cache, 8 MB shared cache, and 8 GB of DDR3-1600 RAM.

For our evaluation, we selected nine popular servers programs on Linux: Apache httpd (v2.2.23, *mpm_worker_module*), nginx (v0.8.54), lighttpd (v1.4.28), mysqld (v5.1.65), Open sshd (v3.5), proftpd (v1.3.3), memcached (v1.4.20), exim (v4.69), and BIND named (v9.9.3).

To benchmark our web servers (httpd, nginx, lighttpd), we relied on the Apache benchmark [1] configured to issue 25,000 requests with 100 concurrent connections and 10 requests/connection. To benchmark mysqld, we relied on the Sysbench OLTP benchmark [6] configured to issue 10,000 transactions using a read-write workload with 100 concurrent connections. To benchmark sshd, we relied on the OpenSSH test suite. To benchmark proftpd, we relied on the pyftpbench benchmark [4] configured to open 100 connections and request 100 1 KB files per connection. To evaluate memcached, we relied on the memslap benchmark [3] configured to issue 1,000,000 operations with a 100 concurrency level. To benchmark exim, we relied on a script repeatedly launching the `sendemail` program [5]. To benchmark named, we relied on queryperf [2] configured to issue 500,000 local requests using 100 concurrent threads. To measure *CodeArmor*'s performance on standard benchmarks, we also considered all the C programs in the SPEC CPU2006 benchmarks.

We compiled all our programs with gcc at -O3, producing PIE x86_64 binaries. We ran all our experiments 11 times—with the CPUs fully saturated throughout our tests—and reported the median.

Our evaluation answers 3 key questions: (i) *Security*: Is *CodeArmor* effective in mitigating both read-based and execution-based disclosure attacks and also countering CRAs that only rely on data disclosure? (ii) *Performance*: Does *CodeArmor* yield acceptable run-time overhead? (iii) *Memory usage*: How much memory does *CodeArmor* use?

8.1. Security against read-based attacks

To measure how well *CodeArmor* defends against read-based disclosure attacks, we look at the attackers' ability to find potential gadgets by reading the code space.

Given an arbitrary read vulnerability, attackers typically use code pointer(s) leaked from the heap or stack to probe as many code pages as possible. However, since *CodeArmor* can leak only code pointers that point to virtual code space and hides the version offset between virtual code space and concrete code space, known-pointer attacks cannot use such pointers to find the concrete code page. This forces the attacker to switch to a blind read-based attack and probe the memory space by brute forcing. With a randomly located concrete code space, the probability of guessing any of its two mapped versions (e.g., `V` and `V-1`) in a 48-bit address space on x86_64 is $2^{-47} \cdot$`sizeof(`*concrete code space*`)`.

We evaluated this probability for both SPEC and our server programs. Results from Figure 7 report an average probability of less than $1.5 \cdot 10^{-7}$ on SPEC and $2.2 \cdot 10^{-7}$ on server programs (geometric mean). Observe that even in the worst case (named), the probability of finding a concrete code space version is less than $6.3 \cdot 10^{-7}$.

Figure 7 shows that the probability of finding a single gadget, while low, depends on the size of a program's memory footprint. Even if the attacker could find a gadget, however, it does not stay valid for long, as *CodeArmor* uses frequent run-time randomization of the concrete code space to invalidate all gadgets found so far. Figure 8 shows that for most server programs (all in a single-threaded configuration), the concrete code space will be typically re-randomized within 2 syscalls. Figure 9, in turn, shows that a server program usually needs around 100 syscalls (and never fewer than 10) to serve a single request. In other words, *CodeArmor* ensures that, during even a single interaction with the server (e.g., a client request), the code space is always re-randomized several times (around 60 times on average, geometric mean). The extremely low re-randomization latency causes all the read-based attacks that require multiple interactions with the server to fail, as all (known and unknown) gadgets will always have moved across requests. Even an attacker who happens to leak the `VERSION_OFFSET(V)` from *Libarmor*

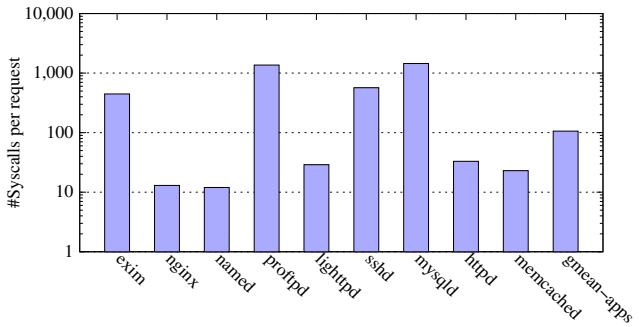


Figure 9. Average number of syscalls to serve a client request

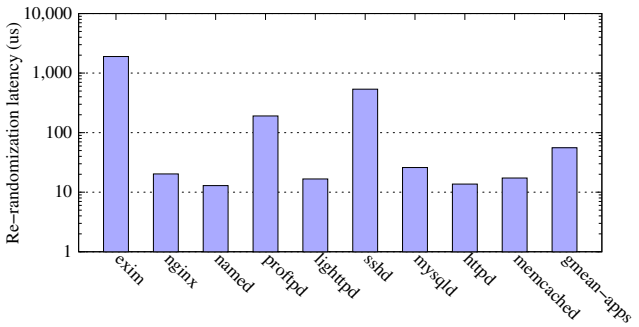


Figure 10. Average server re-randomization latency (μs)

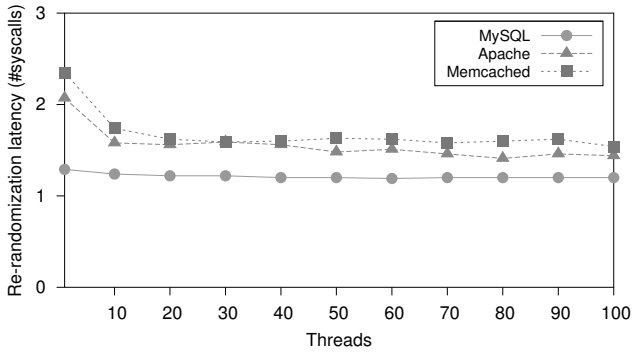


Figure 11. Average server re-randomization latency vs. threads (#syscalls)

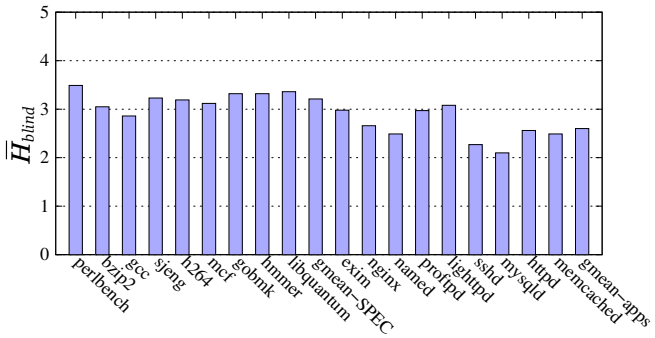


Figure 12. Average number of honey gadgets in a blind attack

data (e.g., using allocation oracles [70] or other side channels [17], [51], [52]), in fact, can gather only obsolete information, given that the offset to the current concrete code space version will always have changed (multiple times) at the next interaction with the server.

Syscall-based re-randomization latency is the most relevant run-time randomization metric for disclosure attacks against servers programs, which inherently need to rely on system calls. For comparison purposes, however, Figure 10 also reports our re-randomization latencies in microseconds. As the figure shows, our design can effectively re-randomize popular server programs at the microsecond granularity (55μs on average, geometric mean). Existing run-time randomization systems only re-randomize at the second [31], [24] or millisecond [32], [92] granularity while often incurring nontrivial performance overhead, making *CodeArmor* the fastest run-time randomization system to date.

To evaluate the effectiveness of run-time randomization on multithreaded applications, we repeated the experiment in Figure 8 using *httpd*, *memcached* and *mysqld* and increasing the number of worker threads up to 100. Figure 11 reports our findings. As shown in the figure, the results reveal very stable behavior across different programs and thread counts. When using 100 worker threads, for example, we observed our three multithreaded programs issuing between 1.2 (*mysqld*) and 1.54 (*memcached*) syscalls for each randomization cycle. The low and stable latency reported confirms the excellent scalability properties of our RCU-based synchronization strategy.

8.2. Security against execution-based attacks

Even without read-based disclosures, attackers may attempt execution-based attacks to infer gadgets by means of crashes, hangs, and other externally observable behavior [15], [78]. Execution-based attacks trying to directly probe the concrete code space, however, will face the same challenges as those mentioned for read-based attacks.

Execution-based attacks on the virtual code space are more interesting, given that they operate similarly to the original program. With *CodeArmor*, however, such attacks are also challenging for two reasons. First, execution-based attacks are limited to legitimate control transfers (thus incurring a slowdown), with *all* the other transfers seamlessly redirected to honey gadgets. Furthermore, since there is no run-time virtual code space randomization, any valid gadget address in the virtual code space remains stable. Constructing a code-reuse chain, however, is still hard, given the second reason: *CodeArmor*'s virtualization and code diversification prevent attackers from easily expanding a set of gadgets from a known pointer, forcing them to resort to blindly probing the virtual code space. Doing so

TABLE 1. NUMBER OF GADGETS AVAILABLE TO ATTACKERS AFTER MEMORY DISCLOSURE: *CodeArmor* (CA) VS. CONTROL-FLOW INTEGRITY (CFI).

	Forward-edge Gadgets						Backward-edge Gadgets						Total Gadgets	
	Binary		Libc		Libraries		Binary		Libc		Libraries			
	CA	CFI	CA	CFI	CA	CFI	CA	CFI	CA	CFI	CA	CFI	CA	CFI
exim	73	2,289	61	2,713	19	6,895	255	9,444	96	8,301	382	14,316	896	43,958
nginx	371	1,770	44	2,713	25	11,255	41	4,890	42	8,301	72	12,625	595	41,554
named	705	6,892	73	2,713	276	20,973	131	44,072	29	8,301	59	34,708	1,273	117,658
proftpd	352	1,704	74	2,713	24	4,475	36	11,528	58	8,301	121	1,135	665	29,856
lighttpd	68	776	52	2,713	24	5,630	34	2,449	35	8,301	67	563	280	2,043
sshd	31	1,227	44	2,713	53	8,212	28	6,448	60	8,301	79	12,593	295	39,494
mysqld	375	11,842	63	2,713	74	7,981	429	42,145	76	8,301	65	8,434	1,082	81,416
httpd	1,378	3,749	57	2,713	37	4,644	74	10,886	64	8,301	87	1,887	1,697	32,180
memcached	9	531	52	2,713	67	3,446	59	1,866	71	8,301	66	3,063	304	19,920
geomean	159	2,170	57	2,713	43	7,050	76	8,802	55	8,301	90	5,130	480	34,166

probabilistically triggers multiple honey gadgets before even a single additional gadget is found.

To estimate the slowdown on a (blind) execution-based attack, we measured the number of control transfers available in the program with and without *CodeArmor*. The results revealed an average reduction for our server programs and SPEC of 71.5% and 76.3% respectively (geometric mean). With *CodeArmor*, all the misaligned control transfers are eliminated. Assuming aligned/misaligned control transfers are uniformly distributed across relevant gadgets, our results suggest a slowdown of up to 4 times on brute-force attacks.

More importantly, given the large number of misaligned control transfers, blind probing of the virtual code space will end up in *CodeArmor*'s honey code space with high probability. Figure 12 shows that (even *before adding random gaps* with our diversification pass) a blind attack triggers roughly around 3 alerts on average before finding a legitimate control transfer. Moreover, these are pessimistic numbers, because in reality the random gaps added by *CodeArmor* all point to honey code space, making it even more likely to trigger alerts. If, instead of probing blindly, the attacker attempts a known-pointer attack, the situation will be even better, since the length of random gaps around any leak-prone pointers guarantees a doubling of the probability of triggering honey gadgets.

8.3. Security against data-only disclosure attacks

The previous sections demonstrated that *CodeArmor* provides strong security guarantees against code disclosure attacks. Armed with data-only disclosure attacks, however, attackers may still leak live code pointers from data memory and attempt to use them "as-is" as gadgets. To examine the residual attack surface, we measured the number of live code pointers found in data memory and directly compared the resulting number of gadgets with those allowed by traditional binary-level CFI solutions based on static analysis [75], [95].

To ensure a fair comparison, we assumed the worst case scenario for our analysis, namely, an attacker able to disclose *all* the code pointers from *any* data region in memory. To simulate this attack scenario for *CodeArmor* and every given server program, we dumped all the possible data regions (i.e., stack, heap, etc.) after completing a full benchmark run (and built-in test suite run, when available) and conservatively scanned the memory dump to enumerate an overapproximation of all the possible code pointers.

To simulate (and even optimistically overapproximate) traditional binary-level CFI solutions [75], [95], in turn, we gathered static analysis-based statistics while restricting (i) forward edges targeting jump tables to the targets already resolved by Dyninst, (ii) forward edges targeting function entry points to the set of functions with address taken (binary and libraries), (iii) backward edges to the set of callsites (binary and libraries). To compute the set of functions with address taken, we implemented a static analysis conservatively scanning code and data for function references, drawing from similar analyses described in prior work [75], [95].

Table 1 presents our findings. The first group of columns reports results for forward-edge (indirect jump/call) gadgets, the second group of columns reports results for backward-edge (return) gadgets, and the last group aggregates the results. We break down our results by target code region (i.e., binary, libc, other libraries). As shown in the table, *CodeArmor* significantly reduces the total number of gadgets available to attackers compared to binary-level CFI, resulting in two orders of magnitude less gadgets on average (480 vs. 34,166).

On the forward edge, the significant reduction is due to two factors. First, unlike CFI, *CodeArmor*'s code pointer table hiding strategy prevents leaking gadgets from jump tables and GOT, resulting in a drastic indirect jump target reduction. Second, CFI's static approach allows all the possible (address-taken) function entry gadgets, which is a vast overapproximation of *CodeArmor*'s live function pointer set. Not surprisingly, the reduction is more impressive for

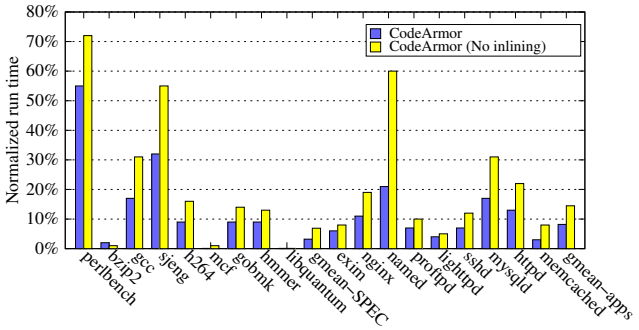


Figure 13. *CodeArmor*'s run-time performance overhead

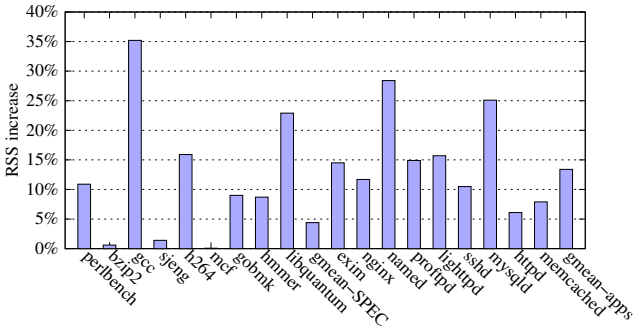


Figure 14. *CodeArmor*'s run-time RSS increase

libc and other libraries—with *CodeArmor* reporting 57 and 43 gadgets (respectively) compared to several thousands for CFI—given that programs tend to cover a small fraction of library code during the execution, increasing the gap with CFI's static approach.

On the backward edge, the even more significant reduction can be explained by the much smaller leakage surface for *CodeArmor*. The program stack is typically short and yields aggressive memory reuse during the execution, limiting the number of return addresses that can be effectively leaked from it by attackers. CFI's static approach, on the other hand, resorts to a vast number of callsites computed by static analysis.

Deploying a shadow stack [33] would, in principle, remove around 65% of the total gadgets for CFI and 46% of the total gadgets for *CodeArmor* (on average), confirming the effectiveness of a shadow stack in mitigating code-reuse attacks [21]. Nevertheless, the security of traditional shadow stack implementations relies on the integrity of (large) ASLR-protected per-thread data regions, which are still accessible to the powerful attacker considered in our threat model using arbitrary memory reads/writes. Finally, we note that, while our analysis focused on overall gadget counts, unlike *CodeArmor*, CFI can enforce separate policies based on the edge type (e.g., callsites cannot be targeted by forward edges). Nevertheless, edge type-based (or even more sophisticated [85]) CFI policies could be easily integrated in *CodeArmor* to further restrict control transfer targets.

8.4. Performance

CodeArmor's performance overhead is mainly due to the cost of frequent control transfer translations. To reduce this cost, *CodeArmor* relies on a binary-level inlining strategy to reevaluate inlining decisions made by the compiler. Figure 13 shows the overhead with and without inlining. Inlining brings down the geometric mean overhead of 6.9% for SPEC and 14.5% for servers to 3.2% for SPEC and 8.2% for servers, yielding a substantial speedup in both cases. This demonstrates *CodeArmor*'s final performance overhead is relatively low and comparable to some of the fastest binary instrumentation solutions available.

8.5. Memory usage

CodeArmor introduces a constant run-time memory usage increase owing to the rewriting of the binary and libraries (and especially the inlining). To evaluate the resulting impact, we measured the maximum memory usage increase (i.e., Resident Set Size or RSS increase) induced by *CodeArmor* during the execution of our benchmarks. Figure 14 reports our findings.

As shown in the figure, *CodeArmor* results in relatively low RSS impact, with an average increase of 13.4% on server programs and 4.4% on SPEC (geometric mean). Even the worst-case RSS impact is realistic, with gcc reporting a 35.2% increase. Our results demonstrate that both our aggressive inlining and re-randomization strategy have very little memory usage impact in practice. Inlining, in particular, has little RSS impact due to demand paging and the good code locality achieved by our cost-effective diversification strategy. Re-randomization, in turn, has essentially no RSS impact, due to our physical memory reuse strategy across concrete code space versions. This demonstrates that *CodeArmor*'s final memory usage increase is realistic, confirming that *CodeArmor* offers a practical solution against diversification-aware code-reuse attacks.

9. Related Work

9.1. Active code-reuse defenses

Active code-reuse defenses seek to actively detect and stop code-reuse attacks. Control-Flow Integrity (CFI) [7], [38], [89], [16], [95], [94], [73] enforces statically extracted invariants to ensure that the execution stays within the boundaries of the original control flow graph (CFG). Strong CFI requires an accurate CFG, which is hard to obtain for binaries. Further, to reduce the overhead, most practical binary-level implementations use coarse-grained CFG invariants, ultimately leaving room to the attackers to launch practical code-reuse attacks [49], [35]. Similar attacks [50], [35], [22] have been recently demonstrated against heuristic-based code-reuse defenses such as KBouncer [73], which relies on the branch history to detect anomalous control flows. More

recent defenses enforce stronger context-sensitive control-flow integrity properties [62], [84], but their effectiveness is subject to the precision of static data-flow analysis. In another direction, recent context-insensitive solutions rely on type inference techniques to improve the precision of binary-level CFI [85], [74]. Multi-variant execution systems [26], [76], [60], [87], [41], [88], [86], [19], finally, can detect any attempts mount a code-reuse attack or even disclose randomized code addresses, at the cost of using additional resources to replicate the execution across variants.

9.2. Code diversification

Code diversification is an instance of fine-grained address space layout randomization (ASLR), a general randomization technique also applied to stack, heap, and static data memory [93], [69], [58], [31], [13], [12]. The goal of code diversification is to keep from the attacker any knowledge of the code and code locations using function/basic-block/instruction permutations [90], [72], [61], [55] or other randomization techniques [31], [28], [25], [10], [34], [56]. Without such knowledge, ROP [79], and other traditional code-reuse attacks [36], [20], [40], [59], [23], become much more complicated. Unfortunately, recent attacks have demonstrated that all these defenses remain vulnerable to information disclosure attacks without adequate protection [82], [15], [78].

9.3. Code disclosure defenses

State-of-the-art code disclosure defenses either focus on designing disclosure-resistant diversification techniques [34], [10] or emulating X⊕R semantics [9], [27], [42], [18], [30], [83], [91], [43], [64] to ensure code is executable but not readable. In either cases, such techniques mainly defend against read-based attacks, while leaving generic binaries vulnerable to execution-based attacks. In addition, many of these techniques assume a much weaker threat model than *CodeArmor*'s—e.g., Isomeron [34] assumes no brute-force attacks, Oxymoron [10] assumes limited code pointer leaks from data memory [34], XnR [9] assumes an attacker unable to leak code pages currently being executed, approaches based on destructive code reads [83], [91] assume gadget reads cannot be used to infer other gadgets [81]. Finally, unlike *CodeArmor*, some of these techniques are not completely self-contained, but instead rely on source code [27], [42], [30], [64], [18], hardware features [27], [42], [43], [91], kernel modules [9], [83], [91], or hypervisors [27], [42], [30], [91].

9.4. Run-time randomization

Similar to *CodeArmor*, run-time randomization solutions draw from live update techniques [44], [46], [53], [48], [45], [47], [67], [66], [65], [8] to periodically re-randomize the memory layout of a running program. This is to either obtain statically sound performance results [32] or counter information leakage [31], [14], [63], [92], [24]. Unlike *CodeArmor*, none of the existing solutions is targeted to very low-latency run-time code space re-randomization. STABILIZER [32] re-randomizes function addresses every 500ms. The system proposed in [31] can periodically re-randomize the entire memory space every few seconds, but incurring a nontrivial overhead at low latencies (up to 50%). Remix [24] provides better performance when randomizing at the second granularity, but can only re-randomize basic blocks within each function. TASR [14] re-randomizes source-level code pointers at every I/O system call, but its *synchronous* re-randomization strategy stalls I/O operations degrading latency (and performance) for applications other than CPU-intensive programs [92]. RUNTIMEASLR [63] re-randomizes the address space for each forked worker process to defeat clone-probing attacks (e.g., BROP attacks). Although RUNTIMEASLR introduces no overhead on forked worker processes, the dynamic pointer tracking used in the parent process introduces high runtime overhead and makes this approach only suitable for server applications with a known (i.e., annotated) process model. In addition, the worker process is still vulnerable to memory disclosure attacks due to the lack of periodic re-randomization. The recent Shuffler [92], finally, can re-randomize binary programs in an egalitarian fashion similar to *CodeArmor*, but requires source-level information and can only re-randomize at the ms granularity. Unlike these systems, *CodeArmor* efficiently re-randomizes the entire concrete code space at very low latencies (μs granularity), relies on a generic and scalable RCU-based synchronization mechanism—STABILIZER [32] relies on traps, Shuffler [92] and Remix [24] on signals, the systems proposed in [31], [63] on system design, and TASR [14] on a kernel module—and implements application-transparent re-randomization eliminating the need for source-level information and annotations. This also makes *CodeArmor* the first generic binary-only run-time re-randomization system to date.

10. Conclusion

We presented *CodeArmor*, a new solution that relies on a virtualized and dynamically randomized code space to thwart modern diversification-aware code-reuse attacks. Specifically, our solution stops all read-based disclosure attacks as code pointers become meaningless to the attacker and the concrete code space is kept out of reach. Execution-based disclosure attacks become exceedingly hard. First, the attacker can use only legitimate control transfers. Second, the virtualized code space allows us to fill most of the memory space with honey gadgets that trigger alerts when targeted, essentially for free. As a result, all attempts to blindly probe memory using executions is likely to trigger (many) alerts.

Our solution works entirely at the binary level with no need for debug symbols, hypervisors, special hardware features, or changes to the operating system. Combined with an average overhead of 6.9% on SPEC (and even lower with aggressive binary-level inlining optimizations), we believe this makes *CodeArmor* practical for many deployment scenarios.

Acknowledgements

We would like to thank the anonymous reviewers for their comments. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571.

References

[1] "Apache benchmark," http://httpd.apache.org/docs/programs/ab.html.

[2] "BIND," http://www.isc.org/downloads/bind.

[3] "memslap," http://docs.libmemcached.org/bin/memslap.html.

[4] "pyftpdlib," https://code.google.com/p/pyftpdlib.

[5] "SendEmail," http://caspian.dotconf.net/menu/Software/SendEmail.

[6] "SysBench," http://sysbench.sourceforge.net.

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS*, 2005.

[8] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: Online patches and updates for security," in *USENIX SEC*, 2005.

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

[10] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *USENIX SEC*, 2014.

[11] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *PASTE*, 2011.

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits," in *USENIX SEC*, 2005.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *USENIX SEC*, 2003.

[14] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *CCS*, 2015.

[15] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *S&P*, 2014.

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

[17] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory deduplication as an advanced exploitation vector," in *S&P*, 2016.

[18] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A. R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *NDSS*, 2016.

[19] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicae for defeating memory error exploits," in *IPCCC*, 2007.

[20] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *CCS*, 2008.

[21] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX SEC*, 2015.

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

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

[24] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *CODASPY*, 2016.

[25] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the tor browser against de-anonymization exploits," in *PETS*, 2016.

[26] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *USENIX SEC*, 2006.

[27] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *S&P*, 2015.

[28] S. Crane, A. Homescu, and P. Larsen, "Code randomization: Haven't we solved this problem yet?" 2016.

[29] S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Booby trapping software," in *NSPW*, 2013.

[30] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a TRaP: Table randomization and protection against function-reuse attacks," in *CCS*, 2015.

[31] A. K. Cristiano Giuffrida and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX SEC*, 2012.

[32] C. Curtsinger and E. D. Berger, "STABILIZER: Statistically sound performance evaluation," in *ASPLOS*, 2013.

[33] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *ASIACCS*, 2015.

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

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

[36] S. Designer, "Return-to-libc attack," BugTraq, August 1997.

[37] M. Desnoyers, P. E. McKenney, A. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Trans. on Parallel and Distributed Systems*, vol. 23, 2012.

[38] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *OSDI*, 2006.

[39] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *S&P*, 2015.

[40] A. Francillon and C. Castelluccia, "Code injection attacks on Harvard-architecture devices," in *CCS*, 2008.

[41] R. Gawlik, P. Koppe, B. Kollenda, A. Pawlowski, B. Garmany, and T. Holz, "Detile: Fine-grained information leak detection in script engines," in *DIMVA*, 2016.

[42] J. Gionta, W. Enck, and P. Larsen, "Preventing kernel code-reuse attacks through disclosure resistant code diversification," 2016.

[43] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *CODASPY*, 2015.

[44] C. Giuffrida and A. Tanenbaum, "Safe and automated state transfer for secure and reliable live update," in *HotSwUp*, 2012.

[45] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum, "Back to the future: Fault-tolerant live update with time-traveling state transfer," in *LISA*, 2013.

[46] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum, "Mutable checkpoint-restart: Automating live update for generic server programs," in *Middleware*, 2014.

[47] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," in *ASPLOS*, 2013.

[48] C. Giuffrida and A. S. Tanenbaum, "A taxonomy of live updates," in *ASCI*, 2010.

[49] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *S&P*, 2014.

[50] E. Goktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *USENIX SEC*, 2014.

[51] E. Goktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining information hiding (and what to do about it)," in *USENIX SEC*, 2016.

[52] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *NDSS*, 2017.

[53] C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C," *TOPLAS*, 2014.

[54] M. Hirzel and A. Diwan, "On the type accuracy of garbage collection," in *ISMM*, 2000.

[55] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *S&P*, 2012.

[56] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *CGO*, 2013.

[57] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *S&P*, 2013.

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

[59] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, "Inspector gadget: Automated extraction of proprietary gadgets from malware binaries," in *S&P*, 2010.

[60] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *DSN*, 2016.

[61] H. Koo and M. Polychronakis, "Juggling the gadgets: Binary-level code randomization using instruction displacement," in *ASIACCS*, 2016.

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

[63] K. Lu, S. Nrnberger, M. Backes, and W. Lee, "How to make ASLR win the clone wars: Runtime re-randomization," in *NDSS*, 2016.

[64] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *CCS*, 2015.

[65] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *USENIX ATC*, 2009.

[66] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," in *PLDI*, 2009.

[67] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *PLDI*, 2006.

[68] B. Niu and G. Tan, "Per-input control-flow integrity," in *CCS*, 2015.

[69] G. Novark and E. D. Berger, "DieHarder: securing the heap," in *CCS*, 2010.

[70] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *USENIX SEC*, 2016.

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

[72] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *S&P*, 2012.

[73] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *USENIX SEC*, 2013.

[74] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, "MARX: Uncovering class hierarchies in C++ programs," in *NDSS*, 2017.

[75] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," 2015.

[76] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space," in *EuroSys*, 2009.

[77] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *S&P*, 2015.

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

[79] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[80] A. Slowinska, T. Stancescu, and H. Bos, "Body armor for binaries: Preventing buffer overflows without recompilation," in *USENIX ATC*, 2012.

[81] K. Snow, R. Rogowski, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *S&P*, 2016.

[82] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, "Just-In-Time code reuse: On the effectiveness of fine-grained address space layout randomization," in *S&P*, 2013.

[83] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *CCS*, 2015.

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

[85] V. van der Veen, E. Göktaş, M. Contag, A. Pawloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *S&P*, 2016.

[86] S. Volckaert, B. Coppens, and B. De Sutter, "Cloning your gadgets: Complete ROP attack immunity with multi-variant execution," 2015.

[87] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz, "Secure and efficient application monitoring and replication," in *USENIX ATC*, 2016.

[88] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "GHUMVEE: Efficient, effective, and flexible replication," in *FPS*, 2012.

[89] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *S&P*, 2010.

[90] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *CCS*, 2012.

[91] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *ASIACCS*, 2016.

[92] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *OSDI*, 2016.

[93] A. S. Xi Chen, H. B. Dennis Andriesse, and C. Giuffrida, "StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries," in *NDSS*, 2015.

[94] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *S&P*, 2013.

[95] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *USENIX SEC*, 2013.