R^2C: AOCR-Resilient Diversity with Reactive and Reflective Camouflage

Felix Berlakovich
μCSRL – Munich Computer Systems Research Lab
Research Institute CODE
University of the Bundeswehr Munich
Neubiberg, Germany
felix.berlakvovich@unibw.de

Stefan Brunthaler
μCSRL – Munich Computer Systems Research Lab
Research Institute CODE
University of the Bundeswehr Munich
Neubiberg, Germany
brunthaler@unibw.de

Abstract

Address-oblivious code reuse, AOCR for short, poses a substantial security risk, as it remains unchallenged. If neglected, adversaries have a reliable way to attack systems, offering an operational and profitable strategy. AOCR's authors conclude that software diversity cannot mitigate AOCR, because it exposes fundamental limits to diversification.

Reactive and reflective camouflage, or R^2C for short, is a full-fledged, LLVM-based defense that thwarts AOCR by combining code and data diversification with reactive capabilities through booby traps. R^2C includes optimizations using AVX2 SIMD instructions, compiles complex real-world software, such as browsers, and offers full support of C++. R^2C thus proves that AOCR poses no fundamental limits to software diversification, but merely indicates that code diversification *without* data diversification is a dead end.

An extensive evaluation along multiple dimensions proves the practicality of R^2C. We evaluate the impact of our defense on performance, and find that R^2C shows low performance impacts on compute-intensive benchmarks (6.6 – 8.5% geometric mean on SPEC CPU 2017). A security evaluation indicates R^2C's resistance against different types of code-reuse attacks.

Keywords: language-based security, software diversity, booby traps, booby-trapped pointers, reactive defenses, code-reuse attacks, address-oblivious code reuse, position-independent code reuse, randomization-based defenses.

ACM Reference Format:
Felix Berlakovich and Stefan Brunthaler. 2023. R^2C: AOCR-Resilient Diversity with Reactive and Reflective Camouflage. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3587439

1 Motivation: Check but not mate!

Over the past decade, we have witnessed a cat-and-mouse game between the ever-increasing sophistication in code reuse attacks, and correspondingly the ever-increasing sophistication in code reuse defenses. Increasingly sophisticated attacks demonstrated shortcomings and invalid defender assumptions, invariably leading to responses by one-upping their defenses. Two defensive research directions were particularly affected: Control-Flow Integrity (CFI) and software diversity. Control-Flow Integrity had some of its assumptions shaken, particularly the idea that it could be fast *and* secure. CFI did eventually see industry adoption through implementations in LLVM, support by Microsoft and by Intel [38, 52, 65]. The demonstrated performance and precision benefits of shadow stacks and their ensuing popularity trace their origins back to this line of work.

Software diversity demonstrated its versatility and resilience by withstanding increasing attack sophistication through building so-called *leakage-resilient software diversification* methods, exemplified by Readactor [25] and Readactor++ [23].

Control-Flow Bending [17] and Control Jujitsu [27] demonstrated the limits of CFI. Address-Oblivious Code Reuse (AOCR), on the other hand, claimed to have identified the fundamental limits of diversity [59]. AOCR's "check-mate" resulted in its call for supporting enforcement-based defenses like Control-Flow Integrity instead. Our research contradicts AOCR's conclusion: specifically, our results provide evidence that AOCR does *not* identify the fundamental limits of diversity; instead AOCR indicates that leakage-resilient diversity focusing exclusively on code diversification is a dead end. AOCR's authors seem to agree with our finding, as they explicitly mention the following:

> Although our variant of code reuse is oblivious to the code layout, it is *not* oblivious to the data layout. In particular, it makes assumptions on the layout of structures as well as the layout of global variables.

Although comprehensive data diversification prevents AOCR in theory, its high performance impact is prohibitive in practice [8, 16]. R^2C's key contribution is a targeted and efficient data diversification technique that focuses on preventing AOCR's inference steps, combined with code diversification to thwart gadget-based code reuse attacks. To increase R²C security, we build on the idea of booby traps to disincentivize brute-force attacks [23–25].

Summing up, this paper contributes the following:

- We present reactive and reflective camouflage, or R²C for short, a new system that combines effective leakage-resilient code diversification with targeted and efficient data diversification. The reactive component frustrates brute-force attacks through pervasive use of booby traps. The reflective component ensures that diversified data is virtually indistinguishable from original data. The camouflage component provides cover for actual code pointers through booby-trapped pointers.
- We discuss the relevant details of a full-fledged, industrial-strength prototype implementation in LLVM.
- We demonstrate an important optimization that uses AVX2 instructions to reduce the performance impact.
- We report the results of our careful and detailed evaluation (see Section 6 and Section 7). Specifically, our evaluation shows:
  - **Performance** We report low performance impacts of 6.6 – 8.5% on the SPEC CPU 2017 benchmark suite (see Section 6.2).
  - **Security** We systematically analyze the security from an attacker's perspective, and find that R²C either thwarts AOCR attacks altogether or reduces attack surface considerably (see Section 7.2).
  - **Scalability** R²C succeeds at compiling complex, real-world software that scales up to 32 million lines of C/C++ code (see Section 6.3).

2 Background

2.1 Evolution of Leakage-Resilient Software Diversity

In 2007, Shacham introduced the first version of return-oriented programming (ROP) attacks [61], a generalization of whole-function reuse attacks [43]. ROP attacks reuse small snippets of assembly that are already present in the target process and end in a free-branch instruction. These snippets are commonly referred to as *gadgets* and by chaining them into a *gadget chain*, an attacker can achieve arbitrary code execution.

ROP attacks typically proceed in three phases:

Several defenses against ROP attacks have been proposed. Broadly speaking, most of these defenses either follow the principle of (i) enforcement, or (ii) randomization. Enforcement-based defenses aim to prevent either the memory corruption, or the gadget chain execution. Examples for enforcement-based approaches are CFI [4], Control-Pointer Integrity (CPI) [44] or Software-based Fault Isolation (SFI) [71].

Randomization-based defenses leverage the observation that ROP attacks depend crucially on the software monoculture. Code-layout randomization techniques, for example, invalidate adversarial assumptions/expectations that all installations of a given piece of software have identical gadget locations [21, 28, 34, 35, 42, 45, 54].

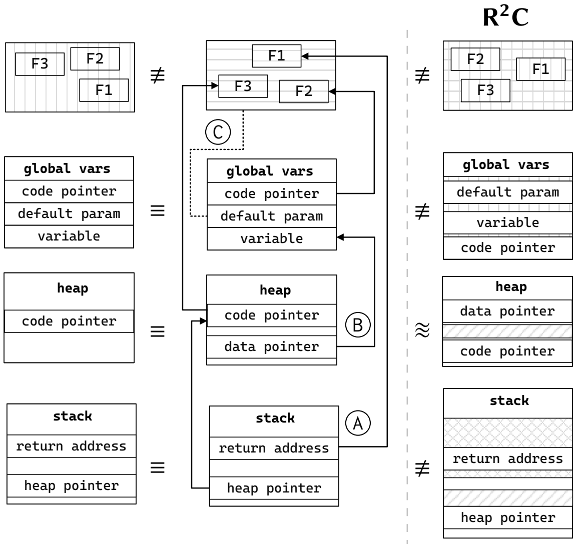


Figure 1. Prior systems primarily diversify code, leaving the layout of observable data predictable (left vs middle, see Section 2.3). R²C (right) diversifies code and observable data.

Code-layout randomization assumes that an attacker cannot disclose the randomized code layout. In 2013, the JIT-ROP attack invalidated this assumption [62]. A JIT-ROP attack proceeds in two stages. First, an attacker reads pointers into the code section to disclose the randomized code-layout at runtime. Second, this learned information is used to relocate a ROP attack to the target process memory layout.

As a response to JIT-ROP, researchers upgraded their defenses to provide *leakage-resilience*. A key component in these defenses is to protect the randomized code with execute-only memory, thereby preventing an attacker from leaking a process' code section.

Unable to disclose the code layout directly, a more sophisticated version of JIT-ROP—*indirect* JIT-ROP—demonstrated the feasibility of inferring gadget locations from code pointers found on the stack [25, 26], which is commonly referred to as *indirect information disclosure.*

2.2 Code-Pointer Hiding

In response to indirect information disclosure, Crane et al. proposed Code-Pointer Hiding (CPH) [25]. To prevent disclosure, CPH redirects code pointers through a randomized trampoline table, located in execute-only memory. AOCR demonstrates that attacks are still possible even in the presence of CPH or similar protection schemes: Even without concrete information about gadgets, CPH function pointers can be called using *whole-function reuse.* Due to the specific layout of CPH's trampoline table, an attacker can further use CPH protected return addresses to reveal the location of such function pointers. Thus, the leakage of return addresses and code pointers remains a threat with or without CPH.

2.3 Address-Oblivious Code Reuse

While indirect JIT-ROP focuses on the discovery of gadgets, address-oblivious code reuse generalizes from sub-function-level granularity to whole-function reuse, such as counterfeit object-oriented programming [60]. As a result of this generalization, defenses focusing on a lower granularity become ineffective.

A closer look at the anatomy of AOCR reveals that it comprises two stages, a profiling stage to harvest valuable code pointers, followed by mounting the actual whole-function reuse attack. AOCR rests on the fact that existing defenses primarily diversify code, but leave the layout of observable *data* intact. Figure 1 shows the information available to an attacker from an unprotected system. Without protection, the global variables, the heap and the stack remain predictable and contain enough information to mount a whole-function reuse attack. Specifically, the attacks in the AOCR paper demonstrate how an attacker can Ⓐ profile pointer locations on the stack, Ⓑ leak heap data to reach the data section, and Ⓒ use the predictable data section layout to corrupt function default parameters (see Figure 2a).

Among the observable data areas, the stack is particularly vulnerable. The stack contains a large number of code pointers (return addresses and function pointers) as well as pointers to the heap and thus serves as a stepping stone to reach other data areas. AOCR's Malicious Thread Blocking further allows an attacker to reliably observe the stack of a single thread.

Existing stack frame diversification techniques, like stack slot randomization, can hinder the exact profiling of function pointer locations [5, 39, 58]. Unfortunately, the location of *return addresses* remains predictable even in the presence of stack slot randomization. The AOCR authors also demonstrate that, unlike for code pointers, an attacker does not need an exact one to one mapping of data pointers to their targets. Instead, a statistical analysis of pointers based on their value ranges can identify *groups of pointers*, such as heap pointers, each of which leads to the desired data area.

3 Threat Model and Assumptions

Our threat model assumes that the attacker has access to a memory corruption vulnerability that enables control-flow hijacking. In particular, we assume that the program contains gadgets for a ROP attack as well as suitable functions for a whole-function reuse attack. In addition, we assume that the attacker can deterministically leak stack frames (e.g., with the help of Malicious Thread Blocking) [59].

Our defense integrates with existing defenses. We assume, specifically, that the data section is protected against code injection (e.g., W⊕X or DEP) [51] and the text section with some form of execute-only memory. Due to compatibility problems with xom-switch, we could not evaluate the impact of execute-only memory in combination with R^{2}C. However, the overhead introduced by execute-only hardware solutions is generally negligible [75].

Our implementation focuses on protecting against information disclosure through the stack or the data section. We do not consider other types of information leaks such as side-channels [40, 41, 48]. Note that using side channels to uncloak EPT-based execute-only memory does not work [32]. Side channels could, however, be used to infer heap information.

4 The Design of R^{2}C

R^{2}C is a leakage-resilient diversity defense against advanced code-reuse attacks. As a first step, and under the protection of execute-only memory, R^{2}C randomizes the order of functions in the text section to thwart ROP and JIT-ROP attacks. R^{2}C further aims to remove the building blocks of sophisticated code-reuse attacks such as indirect JIT-ROP and AOCR:

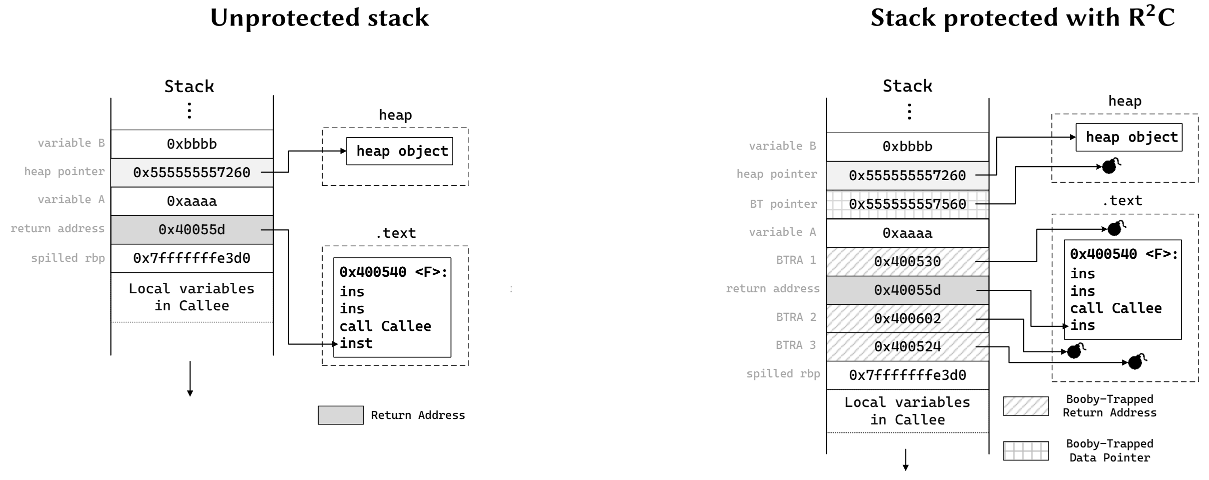
(i) a predictable layout of global variables as a means to manipulate function arguments in a whole-function reuse attack;
(ii) the stack as a source for code and heap pointer leaks.

Similar to Readactor++, R^{2}C protects global variables by randomizing their order and by inserting random padding [23].

To illustrate the challenges of protecting the stack, consider the unprotected stack frame in Figure 2a. Figure 2a shows the stack frame of a function when called *from* function `F` on the x86_64 platform with the System V ABI calling convention. When locating pointers on the stack, an attacker can tap two sources of information. First, an attacker can leverage the predictable location of pointers relative to other stack objects. If the adversary has information pertaining to some of these stack objects (e.g., the value `0xaaaa` of a local variable `A` in function `F`), he can use this information as anchor to locate the pointer. For example, in Figure 2a the return address is one machine word below function variable `A` and adjacent to the spilled base pointer (`rbp`). Second, an attacker can locate pointers based on their value ranges. The heap pointer in Figure 2a, for example, has a different address range than code pointers or the spilled base pointer. If both of these methods fail, an attacker can still resort to guessing. Under specific circumstances, such brute-force attacks are a reasonable option, e.g., some servers restart crashed worker processes without reloading their binary code images (e.g., nginx, Apache, OpenSSH) [11].

To address the described issues, R^{2}C combines two different techniques:

(i) So-called *booby-trapped return addresses*, or BTRAs for short, randomize the precise position of a return



(a) In an unprotected stack frame the return address is at a predictable location surrounded by known values.

(b) R^2C inserts Booby-Trapped Return Addresses to conceal the return address and thus protect it from disclosure.

Figure 2. The stack layout of an unprotected stack (*left*) and a stack protected by R^2C (*right*). We assume the System V ABI on the x86_64 platform.

address in a stack frame. By inserting BTRAs, R^2C changes the relative position of the return address to other stack objects. An attacker is, therefore, no longer able to rely on specific stack objects as anchor points to locate the return address. In addition, BTRAs disguise the return address among enclosing and similar-looking values, as they are specifically chosen to look and behave exactly like benign return addresses. BTRAs point to booby-trap functions randomly distributed in the text section. Without exact code-layout information, an adversary cannot separate BTRA addresses from benign addresses.

(ii) A combination of *booby-trapped data pointers*, or BTDPs for short, and stack slot randomization protects heap pointers against disclosure. Stack object permutation randomizes the position of heap pointers relative to other stack objects. The insertion of BTDPs misleads AOCR's statistical analysis based on pointer value ranges. As BTDPs point into the heap like benign heap pointers, both share the same value range. To protect against brute force attacks, BTDPs point into guard pages. Dereferencing a BTDP causes an immediate fault, giving defenders a way to respond to an ongoing attack.

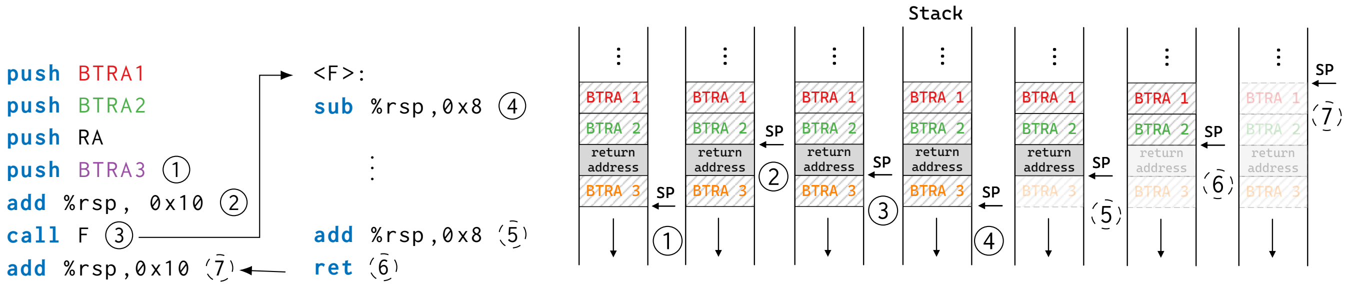
The following subsections give an in-detail presentation of the relevant design decisions and requirements. First, we describe the details on inserting booby-trapped return addresses (see Section 4.1). Second, we explain how BTDPs thwart the localization of heap pointers (see Section 4.2). Third, we illustrate how additional code randomization strengthens the protection afforded by BTRAs and BTDPs (see Section 4.3).

4.1 The Mimicry of Booby-Trapped Return Addresses

An indispensable prerequisite for effective BTRAs is that they must be virtually indistinguishable from actual, benign return addresses. For BTRAs to masquerade as return addresses, they must: (i) look like return addresses; (ii) behave like return addresses; (iii) resist brute force attacks.

To achieve the first and the third goal, BTRAs point to booby trap functions. Booby trap functions are distributed randomly in the *text section*, giving BTRAs the same value range as benign return addresses. Absent exact information about return addresses and leakage through side channels, an attacker could still apply brute-force. Blind ROP, for example, demonstrates the effectiveness and feasibility of brute-force to learn the location of a `read` gadget [11]. Booby traps provide an effective way to penalize such brute force attempts [24]. In the context of R^2C, a classic Blind ROP attack is infeasible for two reasons: (i) the booby trap functions distributed in the text section deter attempts to blindly locate gadgets with brute force; (ii) execute-only memory prevents Blind ROP from disclosing the text section with a found `read` gadget. Likewise, brute forcing an attack with all return address candidates is improbable because all but one of the candidates lead to a booby trap function. We discuss the combination of Blind ROP with the more powerful PIROP attack in Section 7.2.5.

To achieve the second goal, BTRAs mimic the runtime behavior of return addresses. Since return addresses are part of the control flow of a program, they show a distinct observable runtime behavior. Recall the following properties of return addresses and call sites, which must also hold for booby-trapped return addresses:



```
push  BTRA1                  <F>:
push  BTRA2                  sub  %rsp,0x8 ④
push  RA
push  BTRA3 ①                       ⋮
add  %rsp,  0x10 ②
call  F ③                    add  %rsp,0x8 ⑤
add  %rsp,0x10 ⑦             ret ⑥
```

(a) Cooperation between caller (*left*) and callee (*right*) to insert BTRAs.

(b) Stack view of the booby-trapped return address setup and teardown process.

Figure 3. Insertion and deletion of booby-trapped return addresses: Code perspective (Figure 3a), and effect on the stack (Figure 3b). For brevity the figure uses only two BTRAs before (BTRA 1 and BTRA 2) and one BTRA after (BTRA 3) the return address.

(A) A return address occurs exactly once in the stack frame;
(B) Multiple invocations of the same call site have the same return address;
(C) Different call sites have different return addresses.

Violating any of these properties might allow an attacker to learn the actual return address. We take the following precautions to preserve the properties. To preserve property (A), R^{2}C ensures that each call site uses the same booby-trapped return address just once. To preserve property (B), R^{2}C does *not* change the set of BTRAs for a call site at run-time. This decision represents a rare case, where *more* dynamism is *less* effective. Consider a single call site with dynamically changing BTRAs: just two observations suffice to identify the return address, as it is the only pointer remaining identical. To preserve property (C), R^{2}C inserts a randomly chosen set of BTRAs at each *call-site*. If, instead, one were to insert them in the *callee*, multiple call sites would have the identical set of BTRAs, with only one difference: the return address.

For this very reason—preserving property (C)—R^{2}C also avoids reusing booby-trapped return addresses between different call sites as much as possible. In particular, an attacker could leak multiple stack frames and look for recurring addresses to identify BTRAs. Due to the ensuing combinatorial explosion, avoiding the reuse of BTRAs between call sites becomes increasingly difficult with an increasing number of call sites. To counter this combinatorial effect, we tolerate occasional reuse and parameterize the maximum number of BTRAs. An attacker would have to leak two specific stack frames reusing the same BTRAs to gain valuable information. Such a leak is unlikely in practice because the BTRAs are distributed randomly over the text section.

4.2 Booby-Trapped Data Pointers

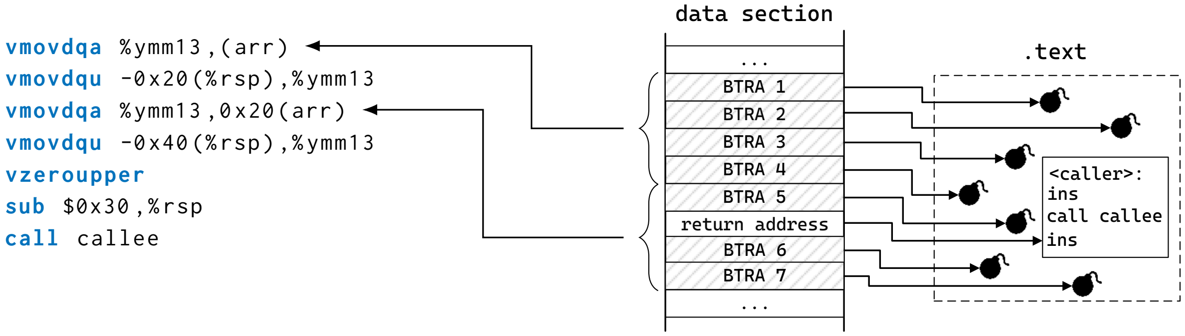
Contrary to return addresses, the runtime requirements for non-control-flow related stack objects are not as strict. Stack objects like local variables, for example, can be permuted freely within the stack frame. Such a reordering invalidates any a priori knowledge an attacker might have regarding the relative position of stack objects to each other.

Without the knowledge of stack object positions an attacker can still resort to analyzing the value ranges of stack objects. The AOCR paper demonstrates that a statistical analysis of two pages of stack values suffices to reliably identify heap pointers. Due to the large address space of x64 systems, the values of pointers occur in clusters, with heap pointers typically constituting the third largest cluster. To reach the heap, an attacker does not necessarily need to identify a *specific* heap pointer. Note, however, that thanks to stack slot randomization an attacker also *cannot* identify a specific heap pointer. Instead, an attacker has to pick and dereference an arbitrary pointer from the cluster of heap pointers. R^{2}C uses this insight to penalize the random choice by mixing BTDPs into the cluster of benign heap pointers. BTDPs point into randomly distributed guard pages on the heap, thus, sharing the same value range as benign heap pointers. A statistical analysis will, thus, sort BTDPs and benign heap pointers into a single cluster. If an attacker dereferences a BTDP, she causes a segmentation fault that can be handled by the program or a monitoring system.

4.3 Strengthening R^{2}C Through Code Randomization

To strengthen security, R^{2}C diversifies the code layout at a sub-function granularity. To this end, R^{2}C randomly inserts NOP instructions at call sites, traps in function prologs, and randomizes register allocation [25, 54]. By diversifying these function parts, two specific types of inference are impeded: (i) from return addresses to function addresses, and (ii) from function addresses to gadget locations.

The inserted NOPs at a call site change the relative offset between the return address and the calling function address. As a result, an attacker can no longer reliably infer the



```
vmovdqa %ymm13,(arr)
vmovdqu -0x20(%rsp),%ymm13
vmovdqa %ymm13,0x20(arr)
vmovdqu -0x40(%rsp),%ymm13
vzeroupper
sub $0x30,%rsp
call callee
```

Figure 4. BTRA setup with AVX2 instructions. The AVX2 instructions load the BTRAs from a call-site specific array in the data section (arr) and write them to the stack in batch.

function address from a leaked return address, effectively restricting the use of return addresses to *gadget localization*. Whereas for a whole-function reuse attack a *single* function pointer might suffice, an attacker typically needs multiple leaked addresses to locate gadgets. Return addresses are protected by BTRAs and increasing the number of required leaks increases the probability of an attacker choosing a BTRA over the real return address. Thus, increasing the number of required leaks increases the overall security.

The inserted traps in the prolog change the relative offset from the function start to a potential gadget location. An attacker leaking a function pointer can, therefore, no longer reliably infer gadget locations and is restricted to *whole-function reuse*. Although in principle, a single leaked function pointer might suffice, whole-function reuse attacks have stricter requirements on the leaked function pointers (e.g., corruptible default parameters). Potential leaks are therefore less likely to meet those requirements. Compared with return addresses, function pointers on the stack occur less frequently and are, furthermore, protected by stack slot randomization.

5 The R^{2}C Compiler

We implemented R^{2}C using the LLVM compiler framework [47]. Our modified compiler supports the compilation of a wide range of systems software for the Linux x86_64 platform (see Section 6.3). To support complex applications like Webkit, R^{2}C is fully compatible with Position Independent Code (PIC) for ASLR, stack unwinding, and exception handling. R^{2}C combines a multitude of different diversification techniques, including function shuffling, global variable shuffling, and NOP insertion. In addition, R^{2}C supports two new diversification techniques that we describe in more detail in the following sections. Specifically, our compiler (i) instruments call sites with booby-trapped return addresses (see Section 5.1); (ii) instruments functions to insert BTDPs, which thwart the statistical analysis of pointers on the stack (see Section 5.2).

5.1 BTRAs: Booby-Trapped Return Addresses

To enclose return addresses with BTRAs, steps ① to ④ in Figure 3 are required. First, the caller pushes randomly chosen BTRAs together with the return address on the stack ①. Note that the position of the return address within the sequence of BTRAs is chosen randomly at compile time. Second, the caller positions the stack pointer above the return address ②. Third, the caller executes the call instruction ③. On x86, call instructions implicitly perform two operations: (i) write the return address to the current stack pointer position; (ii) transfer control to the callee. Due to the prior positioning of the stack pointer, the call instruction overwrites the return address already on the stack. Put differently, the addresses on the stack *do not change* after step ①. Writing all the addresses to the stack at once eliminates the possibility for a race condition window during which an attacker could observe stack changes. If, for example, only the BTRAs were inserted before the call, the attacker could learn the return address by observing the stack right before and after the call instruction [56]. While such a precise timing might seem unattainable in practice, a similar race-condition attack forced Microsoft to rethink their Return Flow Guard architecture [9].

The position of the return address chosen by the *caller* defines how many BTRAs *precede* the return address. The preceding BTRAs create an offset between the return address and caller stack objects (e.g., local variables in the caller). We call the resulting offset *pre-offset*.

After the control-flow transfer, the stack pointer points directly below the return address, still pointing into the BTRA sequence. Subsequent register spills in the callee would, therefore, overwrite BTRAs below the return address. To avoid overwriting the BTRAs, in the last step the callee decreases the stack pointer by a random offset ④. The random offset chosen by the *callee* defines the number of BTRAs that *succeed* the return address. The succeeding BTRAs create an offset between the return address and callee stack objects (e.g., spilled registers). This resulting offset is called a *post-offset*.

After adjusting the stack pointer, the callee executes the regular function prologue. When reaching the epilogue of the callee, the setup process is reversed as shown in Figure 3, steps ⟨5⟩ to ⟨7⟩. First, to identify the correct return address, the callee reverts the post-offset ⟨1⟩ before executing the return instruction. Next, the return instruction pops the return address from the stack ⟨2⟩ and transfers control back to the caller. Finally, the caller reverts the pre-offset ⟨3⟩, which is required to perform stack-relative operations, such as referencing spilled local variables or function parameters.

Since the caller chooses the *number* of BTRAs to push and the *pre-offset*, while the callee chooses the *post-offset*, caller and callee cooperate in the setup of BTRAs. For direct call sites, R^{2}C bounds the number of BTRAs after the return address at compile-time to fit into the *post-offset*. For indirect call sites, no synchronization between the caller and the callee is possible at compile time. Instead, we tolerate that the caller potentially overwrites BTRAs after the return address. Per default, R^{2}C does not add BTRAs to call sites that call unprotected code. While adding BTRAs would not harm functionality, unprotected callees would overwrite *all* BTRAs after the return address. We discuss the security implications in Section 7.4.1.

Depending on the relocation model, the addresses are either embedded in the push instructions or read from the Global Offset Table (GOT). Since an attacker cannot predictably identify the return address among the BTRAs, storing the addresses in the data section does not compromise security.

If the randomly chosen number of BTRAs before the return address is odd, R^{2}C inserts an additional BTRA to keep the stack aligned. On x86-64, the stack pointer must be 16 byte aligned, as programs crash when certain instructions access a misaligned stack.

5.1.1 Stack Arguments.

The BTRAs inserted by R^{2}C change the distance between stack objects above and below the return address. In particular, the distance increases by the sum of pre- and post-offset. This increased distance typically does not cause any compatibility issues, because most functions do not access stack objects above their return address. With one notable exception: In the System V ABI a caller must pass arguments that do not fit into parameter registers on the stack, *above* the return address.

To access stack arguments despite the varying distance, we devised a method called *offset-invariant addressing*. Offset-invariant addressing moves the setup of the frame pointer from the function prologue to the call site. In particular, the caller positions the frame pointer *before* the varying pre-offset. Thus, the distance between the frame pointer and the stack arguments can be computed statically. In the callee we omit the frame pointer setup, and use the already prepared frame pointer to access stack arguments.

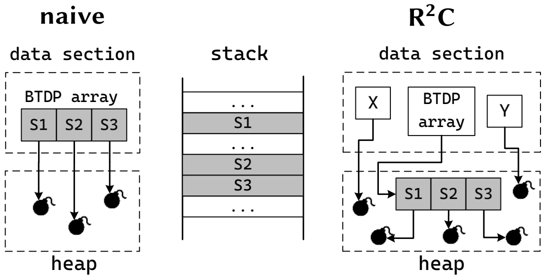


Figure 5. An attacker could observe the BTDPs S1, S2 and S3 occurring in the data section *and* on the stack (left, naive). In R^{2}C no single BTDP occurs in both places (right). The BTDPs X and Y mislead attempts to disclose the BTDP array pointer from the data section (see Section 5.2).

5.1.2 Optimization with AVX2.

Implementing the BTRA setup with push instructions is conceptually straightforward, but exerts significant pressure on the instruction cache. To improve R^{2}C's performance, we built an optimized variant that sets up the BTRAs and the return address with AVX2 vector instructions (see Figure 4) [37].

Before each call site, we initialize a vector register with BTRAs and the return address and write them to the stack. The addresses are read from a call-site specific array in the data section, prepared at compile time. Regarding the security of reading the addresses from the data section, the same reason as for the GOT holds. After the BTRAs and the return address are written to the stack, R^{2}C positions the stack pointer before the return address. Without vzeroupper we observed a performance impact of up to 50%. We discuss the overall performance improvements afforded by the AVX2 instructions in Section 6.2.

5.2 BTDPs: Booby-Trap Data Pointers

BTDPs aim to stop an attacker from following heap pointers found on the stack. To maintain the illusion of a benign heap pointer, BTDPs must have the same value range as real heap pointers, but at the same time cause a fault when being dereferenced. Ideally, BTDPs would point to random addresses on the heap that are protected by memory permissions. Unfortunately, the Intel architecture currently does not offer sub-page level permissions. We therefore simulate the desired effect by letting BTDPs point to random offsets in *guard pages* allocated from heap memory.

Since heap memory is managed by the standard library (e.g. glibc), these allocations cannot be performed at compile time. Instead, R^{2}C registers a *constructor function* that performs the allocations at program start. Specifically, the constructor function allocates a configurable number of page-aligned and page-sized chunks of heap memory. Next, the constructor function frees all but a randomly chosen subset of those allocations. The remaining allocations are scattered randomly across the heap, are page aligned, and span an entire page. R^{2}C stores pointers to random offsets within those allocations in a global pointer array in the data section. To protect the newly created heap pointers from dereferencing, the constructor function revokes the read permission from the occupied page. Allocating the guard pages with `malloc` without freeing them, prevents glibc from reusing the protected page for other allocations.

During compilation, R^{2}C instruments functions to write BTDPs from the pointer array to the stack. How many BTDPs are written per function is chosen randomly chosen using compile-time parameters. The stack slots for the BTDPs are allocated like stack slots for local variables. As a result, stack slot randomization shuffles BTDPs with other stack objects, including benign heap pointers.

Storing the BTDPs in the data section potentially poses a security risk. If an attacker has access to the data section (i.e., knows its location), she could compare heap pointers found in the data section with pointers observed on the stack (see Figure 5). Pointers observed in both locations could potentially be BTDPs. To avoid the risk of triggering a BTDP, an attacker could limit himself to pointers occurring on the stack only. To protect BTDPs against such an attack scenario, R^{2}C applies R^{2}C's principle also to the array holding the BTDPs. Specifically, R^{2}C allocates the pointer array on the *heap* and stores only a *pointer to the array* in the data section. That is, instead of containing the BTDPs directly, the data section now contains only a single heap pointer (apart from application specific pointers). Since an attacker might still be able to locate the pointer to the pointer array, R^{2}C inserts additional BTDPs into the data section. Note, that these additional BTDPs never occur on the stack and the pointers on the heap are inaccessible to the attacker. The attacker thus loses the ability to reliably identify BTDPs. See Figure 5 for a comparison of a naive and a hardened implementation.

As a simple optimization we omit the instrumentation for all functions without stack allocations. Such functions are guaranteed to not write benign heap pointers to the stack either. While this optimization over-approximates the set of functions to instrument, it still improves performance for simple functions (e.g., accessors).

6 Evaluation

6.1 System Configuration

We evaluate R^{2}C on four different machines. Machine EPYC Rome is equipped with an AMD EPYC Rome 7H12 CPU running at 3.2 GHz, 1TB DDR4 RAM running at 3200 MHz, and Debian 11. Machine i9-9900K is equipped with an Intel Core i9-9900K CPU running at 3.6 GHz, 64GB DDR4 RAM running at 2667 MHz, and Debian 11. Machine TR 3970X is equipped with an AMD Ryzen Threadripper 3970X CPU running at 3.7 GHz, 128GB DDR4 RAM running at 2400 MHz, and Debian 10. Machine Xeon is equipped with an Intel Xeon Platinum 8358 CPU running at 2.60GHz, 256GB DDR4 RAM running at 3200 MHz, and Debian 11.

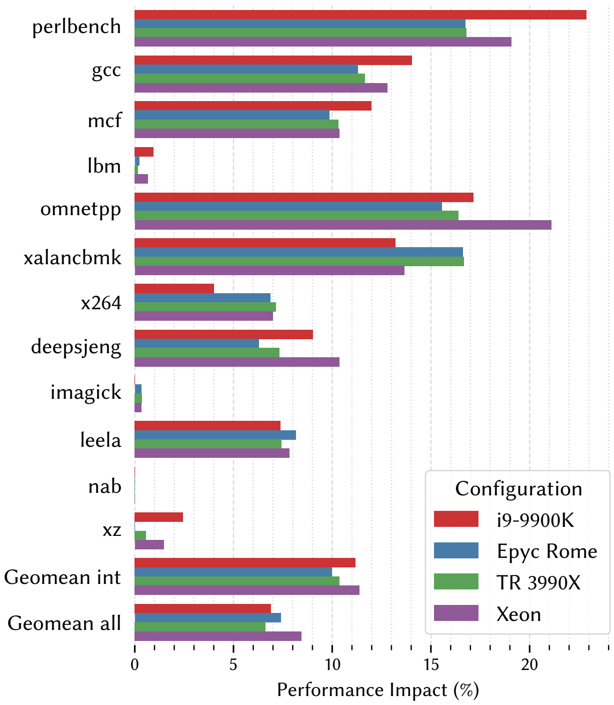


Figure 6. The performance impact of full protection with R^{2}C on four different machines (see Section 6.2.4).

On each machine we used the bundled GCC and gold linker version to compile LLVM. Our LLVM modifications are based on LLVM 11 and we compiled the benchmarks against the bundled glibc and libstdc++ versions.

6.2 Performance

To evaluate the performance impact of R^{2}C, we built and ran the SPEC CPU 2017 benchmark suite using our R^{2}C compiler. The SPEC CPU 2017 suite is a collection of CPU-intensive C and C++ benchmark programs. To allow for direct comparison with prior work, we included the floating point benchmarks [66].

We compiled all benchmarks with the `-O3` optimization level and link-time optimization—LLVM's ThinLTO model in our case—enabled. As LLVM does not yet support the compilation of glibc, we compiled the benchmarks against the unprotected system version of glibc and libstdc++. To measure the worst-case overhead, we also enabled BTRAs for call sites to unprotected code (see Section 7.4.1). For the evaluation of full R^{2}C we took the median execution time of 20 runs. For the analysis of R^{2}C's components we used EPYC Rome and took the median execution time of 12 runs. Since the location of return addresses and the distribution of BTDPs is random, we recompiled the benchmarks with a different seed for each of the executions. To guarantee a fair comparison, we compiled the baseline with the same compiler version and flags but with R^{2}C disabled.

For the webserver benchmarks, we used wrk as client and nginx version 1.14.2 and Apache version 2.4.54, serving 64-byte pages. We split the CPU cores between wrk and the webserver and gradually increased the number of concurrent connections until the CPU was fully saturated. We compared the median throughput of five runs at the previously determined saturation connection count.

6.2.1 BTRAs. We evaluated the overhead of BTRAs with the push setup and with our optimized AVX2 setup sequence respectively. To isolate the overhead of BTRAs, we disabled other diversification measures. We configured R^{2}C to instrument each call site with a total of 10 BTRAs and between 1 and 9 NOPs (see Section 4.3).

For the push setup, 10 BTRAs mean that R^{2}C inserts up to 12 push instructions per call site: 10 for the BTRAs, one for the return address, and one to keep the stack aligned (see Section 5.1). Table 1 shows that the geometric mean overhead of this configuration is 6%, but the outlier omnetpp has an overhead of 21%.

In contrast to the push instructions, setting up 10 BTRAs with AVX2 instructions requires only 7 instructions (see Section 5.1). Table 1 shows that the optimization improves the overall performance by 2%. Most importantly, the optimization decreases the overhead of the outlier omnetpp by about 13 absolute percent points, down to 8%. In this configuration, the maximum overhead of 10% is caused by xalancbmk.

To analyze the overhead of offset-invariant addressing (see Section 5.1.1), we built a configuration without applying any diversification measure, but with offset-invariant addressing *enabled*. Enabling only offset-invariant addressing allows us to measure its performance impact, and the missed opportunities of the frame-pointer omission optimization. We found that the resulting geometric mean performance overhead is 0.79% with a maximum impact of 3.61%. These numbers suggest that the majority of the overhead is caused by writing the BTRAs to the stack.

6.2.2 BTDPs. We configured R^{2}C to insert between zero and five BTDPs per function, but disabled other diversification measures. Table 1 shows that the geometric mean overhead of BTDPs is 2% with xalancbmk causing the maximum overhead of 5%. The optimization to insert BTDPs only in functions that write to their stack frame improves performance by 1%.

6.2.3 Prolog & Layout Randomization. We also isolated the performance impact of prolog trap insertion, and code- and data-layout randomization techniques—i.e., stack slot shuffling, global variable shuffling, and register-allocation randomization. The prolog insertion randomly inserts between one and five traps into each function prolog, causing a geometric mean overhead of 2%, with xalancbmk being most affected at 6%. The combination of layout randomization techniques generally caused negligible overhead.

	max	geomean
Push	1.21	1.06
AVX	1.10	1.04
BTDP	1.05	1.02
Prolog	1.06	1.02
Layout	1.02	1.00

Table 1. The maximum and geometric mean overhead of R^{2}C's components. See Section 6.2.1, Section 6.2.2 and Section 6.2.3 for details. The overhead is relative to the baseline without R^{2}C.

6.2.4 Full R^{2}C. We built a configuration with all R^{2}C protections enabled (see Figure 6). The geometric mean overhead is similar on all systems, with the Xeon machine showing the highest overhead at 8.5% for the full benchmark suite. Some benchmarks show diverging results on different machines. On i9-9900K, perlbench has a significantly higher overhead than on the other machines. For omnetpp, the Xeon machine has the highest overhead at 21%. Conversely, xalancbmk shows better results on i9-9900K and Xeon than on the AMD machines.

On i9-9900K, we found the webserver throughput *decrease* to be 13% for nginx and 12% for Apache. On the AMD machines, the throughput decrease was between 3 and 4 percent for both nginx and Apache.

6.2.5 Memory overhead. To evaluate R^{2}C's memory overhead we linked the benchmark programs from the SPEC CPU 2017 suite to a static library that prints the maxrss rusage metric once the program ends. The maxrss metric is the maximum resident-set size of a process during its lifetime. We chose this methodology because it allows measuring the memory overhead without impacting the benchmark performance. With this methodology, we found the memory overhead of the SPEC benchmarks to be 1-3%.

For the webserver benchmarks we had to choose a different methodology because the webservers spawn child processes. With child processes, maxrss reflects the maximum usage among all child processes instead of the combined maximum usage. Instead, we started a separate monitoring process that records the RSS usage of each webserver process every second and calculated the median. With this methodology, we found the memory overhead of the webserver benchmarks to be about 100%. We verified experimentally

Benchmark	Call Frequency
perlbench	9,435,182,963
gcc	7,471,474,392
mcf	38,657,893,688
lbm	20,906,700
omnetpp	23,536,583,520
xalancbmk	12,430,137,048
x264	3,400,115,007
deepsjeng	11,366,032,234
imagick	10,441,212,712
leela	13,108,456,661
nab	135,237,228,510
xz	3,287,645,643

Table 2. Median call frequencies of SPEC CPU 2017 benchmarks across all inputs.

that about 55% of the memory overhead is caused by the page allocations for BTDPs. The rest is caused by BTRAs and the increased binary size.

6.3 Scalability

Although the SPEC benchmark suite covers a wide variety of test programs, we also compiled real-world software with R^{2}C. Apart from Apache and nginx, we also compiled the GTK version of WebKit [2] and Chromium [1]. We built both browsers with a fixed total number of 10 BTRAs per call site. WebKit and Chromium are massive C++ projects with more than 4.5 million lines and almost 32 million lines of C/C++ code, respectively.

To verify that R^{2}C does not introduce errors into the browser, we ran the included tests as well as the Speedometer browser benchmark. To pass the tests, we had to modify a single source file in Chromium, and three source files in Webkit to deactivate R^{2}C for a few functions. In both cases, unprotected code called an R^{2}C compiled function with stack arguments. We discuss this implementation limitation in Section 7.4.2. We did not include the Speedometer performance results in the performance evaluation since Speedometer's results showed a variation of more than 20% even in the baseline. In daily browsing we did not notice any difference.

7 Discussion

7.1 Performance

As evidenced by the benchmarks, the optimized BTRA setup sequence improves R^{2}C's performance considerably (see Table 1). While our implementation uses AVX2 vector instructions, falling back to SSE vector instructions would be an alternative for more feature constrained CPUs. For CPU's without vector extensions the push based setup sequence provides a viable alternative without loss of security. We ran our benchmarks also on a machine with AVX512 and found that the performance is roughly identical with the same number of vector moves. On such machines, we could either half the BTRA performance impact, or use twice as many BTRAs.

Figure 6 shows that benchmarks with a large number of functions and function calls are affected most by R^{2}C. R^{2}C adds BTRAs *per call site*, explaining the overhead for function heavy benchmarks. To test this correlation with call frequency, we instrumented the SPEC CPU benchmark programs to count the number of executed call instructions. Our instrumentation ignores tail calls because tail calls do not push a return address to the stack and, thus, no BTRAs are inserted. Table 2 shows the median number of calls performed by the SPEC CPU 2017 benchmarks. For each benchmark we took the median call frequencies across all inputs. The data suggests that there is a correlation with the overhead, but this correlation is insufficient to predict the overhead: Perlbench, for example, has less than half the number of calls as omnetpp, but shows a similar overhead.

The difference between the push and AVX2 setup sequence (see Table 1) indicates that increased instruction cache pressure contributes to the overhead. Similarly, prolog trap insertion also contributes to the increased instruction cache pressure.

Surprised by the difference of memory overhead between SPEC CPU 2017 benchmarks and webserver benchmarks, we verified the memory SPEC results by applying the same methodology as for the webserver benchmarks. Instead of relying on the maxrss counter, we recorded the RSS usage of the SPEC benchmarks with a separate monitoring process. The results confirmed a memory overhead of only a few percent. We suspect that for the SPEC benchmarks, memory overhead caused by R^{2}C is low compared to the memory consumed by the benchmark itself. Further research is necessary to substantiate these suspicions.

7.2 Security

While execute-only memory and function shuffling defeat classic ROP and JIT-ROP attacks, indirect JIT-ROP and AOCR remain an issue. For an indirect JIT-ROP attack, an attacker needs to locate valid code pointers, such as return addresses, in readable memory and infer gadget locations based on the found pointers. For an AOCR attack an attacker needs to locate function pointers or infer them from other code pointers (e.g., return addresses), as well as manipulate function parameters. In the following subsections we discuss how R^{2}C counters each of these attack vectors. We also discuss the security of stack unwinding tables and the possibility of an attacker corrupting entire or partial code pointers.

7.2.1 Return Addresses. R^{2}C protects return addresses with BTRAs. As detailed in Section 4.1, the only way for an attacker to identify a return address among the BTRAs is by applying brute force. An attacker's chance to correctly guess the return address depends on the number of BTRAs used. If R is the number of BTRAs for a call site, the probability of correctly guessing the return address is given by $\frac{1}{R+1}$. R^2C's additional code randomization (see Section 4.3) means that an attacker cannot reliably infer the address of the *calling* function based on a leaked return address. As a result, leaked return addresses are only useful to locate gadgets for a ROP chain. When using n return addresses to construct a ROP chain, the success probability for locating all n return addresses decreases to $(\frac{1}{R+1})^n$. For example, with ten BTRAs the probability of successfully finding four return addresses is $(\frac{1}{11})^4 \approx 0.00007$.

7.2.2 Function Pointers. R^2C protects function pointers on the stack and in the data section with stack slot shuffling and global variable shuffling, respectively. R^2C's additional code randomization (see Section 4.3) ensures that such pointers cannot be used to locate gadgets, effectively forcing the attacker into a more constrained whole-function reuse setting. An attacker can learn function pointers from either the stack, the heap, or the data section. Due to ASLR the location of the data section is unknown to the attacker a priori. Leaking a function pointer from the data section, therefore, requires a leaked data section pointer first. Even if an attacker manages to disclose a function pointer that satisfies the criteria for whole-function reuse, global variable shuffling frustrates efforts to locate and manipulate AOCR's default function parameters.

7.2.3 Leaking Heap Data. The attacker can try to leak data from the heap to either learn function pointers or pointers leading to the data section. As ASLR randomizes the heap's location, leaking data requires either a heap pointer or a heap over-read vulnerability. Although R^2C does not protect against leaks from a heap over-read, it does, however, impede the disclosure of heap pointers with BTDPs. The probability of correctly guessing a benign heap pointer among all heap pointers depends on the number of benign heap pointers H, and the number of BTDPs B per function. The probability of randomly picking a benign pointer is $\frac{H}{B+H}$. The exact number for H is application specific and depends, for example, on the number of registers containing heap pointers that are spilled to the stack. B on the other hand is a random variable with a uniform distribution that depends on R^2C's parameters (see Section 5.2). With an expected value of $E(B)$, each stack frame will contain $E(B)$ BTDPs on average. A leak of S stack frames, thus, contains $B = E(X) * S$ BTDPs. BTDPs also increase the risk of detection for attackers trying to locate the heap through random memory probes.

Alternatively, an attacker could try to identify events where BTDPs do not mimic their benign counterparts accurately. For example, by performing heap feng shui an attacker might be able to identify benign heap pointers with a known distance to each other [63]. Note, however, that such an attack requires specific prerequisites and goes significantly beyond the analysis steps of the demonstrated AOCR attacks.

Even if an attacker achieves a heap leak through a heap pointer or a heap over-read, we believe that *leveraging* a heap leak is challenging, considering that (i) scanning large contiguous areas of the heap might hit one of the guard pages used for BTDPs (see Section 5.2); (ii) finding suitable function pointers for a whole-function reuse attack within the leaked window is difficult (see Section 7.2.2).

7.2.4 Exception handling and stack unwinding. As part of the BTRA setup and teardown code, R^2C also emits the necessary `CFI` directives to support exception handling and stack unwinding. `CFI` directives record stack pointer and frame modifications in the `.eh_frame` section. Entries in the `.eh_frame` are not, however, associated with function symbols, but with Program Counter (PC) *ranges*. These PC ranges are unknown to the attacker due to code layout randomization. An attacker cannot, therefore, associate entries in the `.eh_frame` table with functions.

The position of an entry in the table—i.e., its row—could provide the attacker with important information. Each entry in the table, reflects the position of a function in a compilation unit. Through function reordering/permutation row-based references become invalid. Since exceptions occur infrequently, one could also use a more expensive protection scheme, such as encryption, to protect these meta-data.

7.2.5 Corrupting code pointers. Code-reuse attacks typically corrupt entire code pointers. An attack called Position-Independent Code Reuse (PIROP) generalizes this principle by corrupting only parts of code pointers and, as a result, is immune to ASLR and page-level randomization [31]. R^2C impedes a PIROP attack in two ways. First, R^2C shuffles functions and randomizes at the sub-function level (see Section 4.3), thus increasing the entropy for PIROP. Second, BTRAs constrain candidate PIROP gadgets that manipulate (partial) return addresses: In the presence of BTRAs a PIROP attack needs to corrupt *all* return addresses, requiring either iterative gadget execution or more complex gadgets.

7.3 Remaining attack surface

At present, R^2C remains susceptible to two types of brute force attacks. In a Blind ROP scenario with restarting worker processes, an attacker could use PIROP to brute force the entropy resulting from R^2C randomization techniques. Similarly, an attacker could use the corruption of potential return addresses as a side channel. For example, by overwriting selected return address candidates with zero and observing whether the process crashes, the attacker could learn the location of the real return address. Both attacks could be prevented by load time re-randomization. R^2C could also deter the corruption of BTRAs by checking a random subset of BTRAs for consistency after the return.

R^{2}C focuses on code-reuse attacks and the leakage of *control-flow* data. Although R^{2}C's layout randomization raises the bar for attackers [36], R^{2}C does not offer the same protection as defenses specialized for data-only attacks [18].

A way to strengthen R^{2}C's security would be to combine it with Multi-Variant Execution Engine (MVEE)s [7, 13, 22, 69, 70]. MVEEs and diversification defenses like R^{2}C naturally complement each other. Considering that R^{2}C diversifies along multiple dimensions, an MVEE would detect data corruption or leakage in one of the variants with high probability.

7.4 Limitations

7.4.1 Coverage. R^{2}C is able to protect only the call-sites and functions it actually compiles. BTRAs for calls to unprotected functions are disabled by default as these functions would overwrite all the BTRAs after the return address. Without BTRAs after the return address, the return address would always be the last address in the list of addresses. Overwritten BTRAs are only an issue, however, if the attacker knows which parts of the program have not been compiled by R^{2}C. Lacking this information, an attacker does not know where the return address is the last address.

7.4.2 Support for stack argument calling convention with non-R^{2}C compiled code. At present, our prototype implementation does not support calling functions with stack arguments from code not compiled by R^{2}C. This incompatibility is due to such functions expecting the caller to prepare a frame pointer to account for the changed calling convention. During our evaluation of R^{2}C, we encountered just three such cases (one in the unit tests of WebKit, one in the XML parser callbacks of WebKit, and one in the regular expression implementation of Chromium). With three cases in 35 million lines of C/C++ code, we conclude that this combination is rare in practice and, thus, opted for disabling the emission of BTRAs for the affected functions. Note that these cases could also be supported by automatically inserting a trampoline for externally visible functions with stack parameters.

8 Related Work

We group the related work into randomization-based defenses and enforcement-based defenses, the two major strains of research against code reuse attacks. Table 3 gives an overview of the most closely related work on randomization-based defenses. Keep in mind, though, that performance comparison between related work is notoriously difficult, considering the vastly different benchmarking methodologies and assumptions.

8.1 Randomization-Based Defenses

Readactor and Readactor++ combine code-pointer hiding with various randomization techniques and deter brute-force attacks with booby traps [23, 25]. Crane et al. give a detailed overview of the possibilities of reactive cyber booby traps [24]. Similar to Readactor and Readactor++, R^{2}C builds on leakage-resilient diversification and booby traps to deter brute-force attacks. Unlike Readactor and Readactor++, however, R^{2}C withstands AOCR by protecting pointers on the stack against profiling. A proposed extension to CPH protects the trampoline table with cookies against AOCR whole-function reuse attacks, but incurs a prohibitive performance overhead [46].

R^{2}C requires a secret code layout, as otherwise an attacker could leak the BTRA setup code. To prevent runtime disclosure of the code layout, execute-only memory presents a practical solution. Execute-only memory implementations range from software emulation [6, 12] over destructive code reads [64, 72] to hardware assisted solutions [12, 25, 30].

In 2017, Pomonis et al. presented a technique called return-address decoys as part of their kernel JIT-ROP defense, kRˆX [56]. As kRˆX and AOCR were published in the same year, kRˆX does not focus on the prevention of AOCR, but the idea behind return-address decoys is similar to BTRAs. Like return address decoys, BTRAs aim to prevent the disclosure of return addresses during a code reuse attack. Unlike return address decoys, however, R^{2}C supports an arbitrary number of BTRAs, thus strengthening the security guarantees against a brute force attack. R^{2}C also is not susceptible to race conditions (see Section 5.1).

An alternative to preventing the disclosure of the code layout is to invalidate attacker observations with re-randomization. TASR [10], Shuffler [73], and CodeArmor [19] follow this approach. With TASR, a kernel module re-randomizes the code layout and updates pointers whenever certain system calls are invoked. While effective in principle, the re-randomization requires a source code analysis that is specific to the C programming language, limiting TASR's scalability. Instead, Shuffler and CodeArmor introduce abstracted code locators (table index and shifted addresses), regularly re-randomize the code layout and translate the code locators at runtime. In principle, Shuffler's and CodeArmor's code locator translation is similar to Readactor's CPH and, thus, susceptible to AOCR. Isomeron also tolerates the disclosure of code, but thwarts code-reuse attacks by randomly switching between two versions of the code on return [26]. Unfortunately, Isomeron's dynamic binary instrumentation incurs a significant performance overhead.

StackArmor is a binary-only defense against various types of stack attacks [20]. To mitigate temporal and spatial attacks on stack objects, StackArmor diversifies the location of stack frames, isolates vulnerable buffers, and zero-initializes stack variables. While StackArmor provides comprehensive protection, it incurs a substantial performance overhead.

Smokestack randomizes the location of stack objects on each function invocation [5]. In contrast to R^{2}C, Smokestack

Defense	SPEC CPU Overhead (%)	C	C++	ROP	JIT-ROP	PIROP	AOCR
CodeArmor [19]	6.9[a,b]	●	◐[1]	●	●	●	○
TASR [10]	2.1[a,b,c]	◐[2]	○	●	●	●	●
StackArmor [20]	28[a,b,d]	●	◐[1]	●	●	●	●
Readactor [25]	6.4[a]	●	◐[1]	●	●	●	○
kRˆX [56]	n/a[4]	●	n/a[4]	●	●	●	◐[3]
R^{2}C	6.6 – 8.5	●	●	●	●	◐[5]	●

a SPEC 2006
b no C++ benchmarks
c comparison w/-0g, perlbench excluded
d measures #cycles
1 no support for exceptions
2 perlbench not working
3 single decoy; no heap pointer protection
4 not comparable (kernel defense)
5 See Section 7.3

Table 3. Comparison of R^{2}C to related diversification techniques.

focuses on data-only attacks and, therefore, does not randomize the return address.

8.2 Enforcement-Based Defenses

An alternative to randomization-based approaches are enforcement-based defenses. A prominent example is CFI [3, 4]. CFI tolerates memory corruptions, but contains the effects of code-reuse attacks by preventing malicious control-flow transfers. Burow et al. give an overview of the design space of CFI [14]. CFI defenses are generally categorized into forward-edge and backward-edge CFI.

Forward-edge CFI relies on the program’s Control-Flow Graph (CFG) to restrict the targets of indirect branches. Statically determining the CFG has theoretical limits and leads to an overapproximation of possible targets [33]. Thus, CFI-based defenses aim to limit the overapproximation, e.g., by combining CFI with code-layout randomization [53], by considering function signatures [68] or by focusing specifically on C++ [29, 57, 74].

Backward-edge CFI seeks to restrict return branches and is typically implemented with shadow stacks. Shadow stacks can be implemented either in software [44, 49, 55] or in hardware[38, 67] Burow et al. explore the tradeoffs of different software shadow stack implementations [15].

CFI generally prevents ROP and JIT-ROP, but its effectiveness against AOCR depends on whether the malicious control-flow transfers are valid in the approximated CFG. Note that R^{2}C and CFI are orthogonal defenses and could in principle strengthen each other.

CCFI enforces CFI and protects against information disclosure at the same time by encrypting and authenticating code pointers [50]. Unfortunately, CCFI incurs prohibitive performance overheads.

9 Conclusions

As indicated at the beginning, the research community has—until now—neglected to address the danger emanating from address-oblivious code reuse (AOCR) attacks.

R^{2}C offers a way out of this impasse, using a methodological and systematic way that comprises insights from the analysis of prior work, novel diversification techniques, and extensive evaluation. Our results provide evidence of R^{2}C’s low performance impact, scalability to compile complex, real-world software exemplified by browsers, and its probabilistic security guarantees.

Considering our results, R^{2}C contradicts the claim that AOCR identifies fundamental limits of software diversity. Instead, we find that AOCR merely identifies the dead end of focusing exclusively on code diversification, and the necessity of data diversification.

10 Acknowledgements

We thank the anonymous reviewers for their feedback. We would also like to thank the artifact evaluation committee members for their patience and support with improving the paper’s artifacts.

The research reported in this paper has been partially funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Digital and Economic Affairs (BMDW), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) within the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

References

[1] 2022. *Chromium.* https://www.chromium.org/

[2] 2022. *WebKit.* https://webkit.org/

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. A Theory of Secure Control Flow. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* Vol. 3785 LNCS. 111–124. https://doi.org/10.1007/11576280_9

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security* 13, 1 (Oct. 2009), 1–40. https://doi.org/10.1145/1609956.1609960

[5] Misiker Tadesse Aga and Todd Austin. 2019. Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 26–36. https://doi.org/10.1109/CGO.2019.8661202

[6] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, USA, 433–447.

[7] Emery D Berger and Benjamin G Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

[8] Sandeep Bhatkar and R. Sekar. 2008. Data space randomization. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 5137 LNCS. Springer, Berlin, Heidelberg, 1–22. https://doi.org/10.1007/978-3-540-70542-0_1

[9] Joe Bialek. 2018. The Evolution of CFI Attacks and Defenses. In *OffensiveCon 2018*.

[10] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, Vol. 2015-Octob. ACM Press, New York, New York, USA, 268–279. https://doi.org/10.1145/2810103.2813691

[11] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 227–242. https://doi.org/10.1109/SP.2014.22

[12] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, Reston, VA. https://doi.org/10.14722/ndss.2016.23364

[13] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. 2007. Diversified process replicæ; for defeating memory error exploits. In *Conference Proceedings of the IEEE International Performance, Computing, and Communications Conference*. 434–441. https://doi.org/10.1109/PCCC.2007.358924

[14] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity. *Comput. Surveys* 50, 1 (April 2017), 1–33. https://doi.org/10.1145/3054924

[15] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999. https://doi.org/10.1109/SP.2019.00076 arXiv:1811.03165

[16] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. 2008. *Data Randomization*. Technical Report.

[17] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium*. 161–176.

[18] Scott A Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS '17*. ACM Press, New York, New York, USA, 193–204. https://doi.org/10.1145/3052973.3052983

[19] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017*. 514–529. https://doi.org/10.1109/EuroSP.2017.17

[20] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries. https://doi.org/10.14722/ndss.2015.23248

[21] Frederick B. Cohen. 1993. Operating system protection through program evolution. *Computers and Security* 12, 6 (Oct. 1993), 565–584. https://doi.org/10.1016/0167-4048(93)90054-9

[22] Benjamin Cox and David Evans. 2006. N-Variant Systems: A Secretless Framework for Security through Diversity. In *15th USENIX Security Symposium (USENIX Security 06)*. USENIX Association, Vancouver, B.C. Canada.

[23] Stephen Crane, Michael Franz, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Bjorn De Sutter. 2015. It's a TRaP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. ACM Press, New York, New York, USA, 243–255. https://doi.org/10.1145/2810103.2813682

[24] Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Booby trapping software. In *Proceedings of the 2013 workshop on New security paradigms workshop - NSPW '13*. ACM Press, New York, New York, USA, 95–106. https://doi.org/10.1145/2535813.2535824

[25] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *2015 IEEE Symposium on Security and Privacy*, Vol. 2015-July. IEEE, 763–780. https://doi.org/10.1109/SP.2015.52

[26] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, Reston, VA. https://doi.org/10.14722/ndss.2015.23262

[27] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, Vol. 2015-Octob. ACM, New York, NY, USA, 901–913. https://doi.org/10.1145/2810103.2813646

[28] Michael Franz. 2010. E unibus pluram. In *Proceedings of the 2010 workshop on New security paradigms - NSPW '10*. ACM Press, New York, New York, USA, 7. https://doi.org/10.1145/1900546.1900550

[29] Robert Gawlik and Thorsten Holz. 2014. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference on - ACSAC '14*. ACM Press, New York, New York, USA, 396–405. https://doi.org/10.1145/2664243.2664249

[30] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *CODASPY 2015 - Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. 325–336. https://doi.org/10.1145/2699026.2699107

[31] Enes Göktaş, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. 2018. Position-Independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 227–242. https://doi.org/10.1109/EuroSP.2018.00024

[32] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 1871–1885. https://doi.org/10.1145/3372297.3417289

[33] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems* 23, 6 (Nov. 2001), 685–746. https://doi.org/10.1145/506315.506316

[34] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd my gadgets go?. In *Proceedings - IEEE Symposium on Security and Privacy*. IEEE, 571–585. https://doi.org/10.1109/SP.2012.39

[35] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. Librando. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. ACM Press, New York,

New York, USA, 993–1004. https://doi.org/10.1145/2508859.2516675

[36] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*. IEEE, 969–986. https://doi.org/10.1109/SP.2016.62

[37] Intel. 2022. *Intel Advanced Vector Extensions.* https://www.intel.com/content/dam/develop/external/us/en/documents/36945

[38] Intel. 2022. *Intel CET.* https://software.intel.com/content/www/us/en/develop/articles/technical-look-control-flow-enforcement-technology.html

[39] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. 2011. Compiler-Generated Software Diversity. In *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer, New York, NY, 77–98. https://doi.org/10.1007/978-1-4614-0977-9_4

[40] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them. *ACM SIGARCH Computer Architecture News* 42, 3 (10 2014), 361–372. https://doi.org/10.1145/2678373.2665726

[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, Vol. 2019-May. IEEE, 1–19. https://doi.org/10.1109/SP.2019.00002

[42] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-Assisted Code Randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, Vol. 2018-May. IEEE, 461–477. https://doi.org/10.1109/SP.2018.00029

[43] Sebastian Krahmer. 2005. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. (2005).

[44] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, Vol. 14. 147–163.

[45] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 276–291. https://doi.org/10.1109/SP.2014.25

[46] Per Larsen and Ahmad-Reza Sadeghi (Eds.). 2018. *The Continuing Arms Race: Code-Reuse Attacks and Defenses.* ACM. https://doi.org/10.1145/3129743

[47] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[48] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990.

[49] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, Vol. 2015-Octob. ACM Press, New York, New York, USA, 280–291. https://doi.org/10.1145/2810103.2813694

[50] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, Vol. 2015-Octob. ACM Press, New York, New York, USA, 941–951. https://doi.org/10.1145/2810103.2813676

[51] Microsoft. 2022. *Data Execution Prevention.* https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

[52] Microsoft. 2022. *Microsoft Control Flow Guard.* https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

[53] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, Reston, VA. https://doi.org/10.14722/ndss.2015.23271

[54] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 601–615. https://doi.org/10.1109/SP.2012.41

[55] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9148. Springer Verlag, 144–164. https://doi.org/10.1007/978-3-319-20550-2_8

[56] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 420–436. https://doi.org/10.1145/3064176.3064216

[57] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, Reston, VA. https://doi.org/10.14722/ndss.2015.23297

[58] Benjamin D. Rodes, Anh Nguyen-Tuong, Jason D. Hiser, John C. Knight, Michele Co, and Jack W. Davidson. 2013. Defense against Stack-Based Attacks Using Speculative Stack Layout Transformation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7687 LNCS. Springer Verlag, 308–313. https://doi.org/10.1007/978-3-642-35632-2_29

[59] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, Reston, VA. https://doi.org/10.14722/ndss.2017.23477

[60] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*, Vol. 2015-July. IEEE, 745–762. https://doi.org/10.1109/SP.2015.51

[61] Hovav Shacham. 2007. The geometry of innocent flesh on the bone. In *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*, Vol. 22. ACM Press, New York, New York, USA, 552. https://doi.org/10.1145/1315245.1315313

[62] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 574–588. https://doi.org/10.1109/SP.2013.45

[63] Alexander Sotirov. 2007. Heap Feng Shui in JavaScript. In *Black Hat Europe*.

[64] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, Vol. 2015-Octob. ACM Press, New York, New York, USA, 256–267. https://doi.org/10.1145/2810103.2813685

[65] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 941–955.

[66] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2019. SoK: Benchmarking Flaws in Systems Security. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 310–325. https://doi.org/10.1109/EuroSP.2019.00031

[67] Victor Van Der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the ACM Conference on Computer and Communications Security*, Vol. 2015-Octob. 927–940. https://doi.org/10.1145/2810103.2813673

[68] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953. https://doi.org/10.1109/SP.2016.60

[69] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and Efficient Application Monitoring and Replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 167–179. https://www.usenix.org/conference/atc16/technical-sessions/presentation/volckaert

[70] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. 2020. Distributed Heterogeneous N-Variant Execution. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 12223 LNCS. Springer, 217–237. https://doi.org/10.1007/978-3-030-52683-2{_}11

[71] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles - SOSP '93 (SOSP '93)*. ACM Press, New York, New York, USA, 203–216. https://doi.org/10.1145/168619.168635

[72] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS '16*. ACM Press, New York, New York, USA, 35–46. https://doi.org/10.1145/2897845.2897891

[73] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 367–382.

[74] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, Reston, VA, 8–11. https://doi.org/10.14722/ndss.2015.23099

[75] Mingwei Zhang and Ravi Sahita. 2018. eXecutable-Only-Memory-Switch (XOM-Switch): Hiding Your Code From Advanced Code Reuse Attacks in One Shot. In *Black Hat Asia Briefings (Black Hat Asia)*.

A Artifact Appendix

A.1 Abstract

R^{2}C is a novel defense against code-reuse attacks such as AOCR, and is based on the LLVM compiler framework. The provided artifacts consist of the source of the compiler as well as automation scripts to (i) build the compiler; (ii) to build and run benchmarks. We also provide instructions on how to use R^{2}C to compile two major web browsers, WebKit and Chrome. Note that we do not include building Chrome as an experiment due to its complexity and high resource demand (see Appendix A.3).

A.2 Description & Requirements

A.2.1 How to access. Our artifact consists of a modified LLVM compiler suite. We based our modifications on the LLVM `release/11.x` branch, commit `1fdec59b`. The full source code repository is available at https://github.com/fberlakovich/r2c-llvm.

In addition, the benchmarking repository https://github.com/fberlakovich/r2c-benchmarking contains automation scripts to

1. fetch and build the compiler;
2. build and run SPEC CPU 2017 benchmarks;
3. build and run webserver tests.

Due to licensing reasons we cannot include the SPEC CPU 2017 benchmark itself.

The benchmarking repository also contains instructions on how to build web browsers with R^{2}C. For the WebKit browser, we provide https://github.com/fberlakovich/r2c-webkit, which contains the sources of the WebKit version used in the paper and a commit containing the required patches (see the Section 6.3 and Section 7.4.2 in the paper for details).

The artifact evaluation was performed with a snapshot of the benchmarking repository identified by DOI 10.5281/zenodo.7728972, which also contains references to DOIs for the source code repository and WebKit repository.

A.2.2 Hardware requirements. To build the compiler and run the benchmark programs (SPEC CPU 2017 and web servers), you need at least 20GiB of disk space and 16GiB of RAM, although we recommend 32GiB of RAM.

To build WebKit, you need at least 30GiB of disk space and 32GiB of RAM, although we recommend 64GiB of RAM. During compilation, more RAM is generally beneficial because R^{2}C is not optimized for compilation speed and also uses LLVM's LTO feature.

A.2.3 Software dependencies. We require the system to run Debian 10 or 11. The dependencies for SPEC CPU 2017 and the web servers are automatically downloaded by our automation scripts.

The web server benchmarks require a password-less SSH login to localhost. We provide an example configuration in the instructions for running the benchmarks.

For the WebKit build a number of Debian packages must be installed. See the WebKit build instructions for the full list.

A.3 Evaluation workflow

A.3.1 Major Claims.

1. Our solution successfully compiles and runs all C and C++ benchmarks of the SPEC CPU 2017 benchmark suite. This claim is proven by experiment `E1`. The result of this experiment (on our servers) is shown in Figure 6.
2. Our solution compiles and runs Apache and nginx. This claim is proven by experiment `E2`. The result of this experiment (on our servers) is described in Section 6.2.4.
3. Our solution compiles and runs the WebKit GTK browser. The built browser successfully executes the Speedometer benchmark. This claim is proven by experiment `E3`. The result of this experiment is described in Section 6.3.

   Note that for the paper we also built the Chrome browser. Unfortunately, the Chrome build is not automated and hard to setup. Apart from the setup, the Chrome build requires resources likely not present in commodity hardware. For those reasons, we do not expect the artifact evaluation committee to repeat the build steps in reasonable time and, consequently, do not include the Chrome build as an experiment. However, we provide build instructions and details on the exact versions in https://github.com/fberlakovich/r2c-benchmarking.

A.3.2 Experiments.

Experiment (E1): *takes about 15 human minutes and 4 compute hours.* In this experiment, the SPEC CPU 2017 benchmarks are compiled once with full R^{2}C protection. Subsequently, a single run of the protected benchmarks is performed. To conduct the experiment, follow the instructions in `README.md` at https://github.com/fberlakovich/r2c-benchmarking. The section "Evaluating functionality" under "Running SPEC benchmarks" contains instructions on how to run a single iteration of the benchmarks with R^{2}C protection. This experiment tests the functionality of R^{2}C. Note that conducting a full performance evaluation with a reasonable number of iterations (e.g., 10 iterations) requires multiple compute days.

Experiment (E2): *takes about 30 human minutes and 1 compute hour.* In this experiment, the webservers Apache and nginx are built with full R^{2}C protection and the throughput and latency at different connection counts is recorded. To conduct the experiment, follow the instructions at https://github.com/fberlakovich/r2c-benchmarking. The section "Running webserver benchmarks" contains instructions on how to run the throughput tests. To only test R^{2}C's functionality, you can skip the baseline run and overhead calculation described in the instructions.

Experiment (E3): *takes about 30 human minutes and 4 compute hours.* In this experiment, the GTK version of the web browser engine WebKit is built. Subsequently, the built MiniBrowser is used to run the Speedometer benchmark. To conduct the experiment, follow the instructions in `README.md` at https://github.com/fberlakovich/r2c-benchmarking. The section "Building WebKit" contains instructions on how to build the WebKit GTK browser. Once the browser is built, use the command `MiniBrowser` https://browserbench.org/Speedometer2.0/ to visit the Speedometer website.