CSDrop: Leveraging Context-Sensitive Decoding to Thwart Return-Oriented Programming

Zhifeng Jiang*

Abstract

The widespread deployment of *Data Execution Prevention* in modern processors and operating systems has made *code injection attacks* significantly harder. In recent years, attackers have resorted to *Return-Oriented Programming* (ROP), a class of memory corruptions that enables malicious computation without injecting any malicious code. ROP is proven to be Turing-complete for multiple Instruction Set Architectures, threatening a wide range of applications.

While existing defense efforts mainly focus on thwarting ROP in the software level, e.g., code instrumentation and system enhancement, this work aims to eliminate the threat with *microarchitecture modification* by proposing **CSDrop**, a CPU plugin that enables context-sensitive decoding for securely backing up and validating return addresses. We have implemented CSDrop in the gem5 simulator, atop which ROP attacks are first issued by the RIPE attack suite to verify the defense capability. We then conduct a comprehensive evaluation with programs from SPEC CPU 2006 to show that the performance overhead is negligible (~0.6%). Finally, we discuss the practicality of such a method, as well as anticipating potential future works.

1 Introduction

Despite decades of research, *buffer overflow* vulnerabilities still form a major chunk of the security loopholes that plague the Internet as of today [2, 6]. Essentially, a buffer overflow occurs when a *out-of-bound* pointer is used to access data in the address space adjacent to the original buffer allocated on the stack or the heap [40]. An evasive direction of exploiting such vulnerabilities is called *return-oriented programming* (ROP) [37], which utilizes buffer overflow to overwrite return addresses stored on the stack, in a way that the control flow is ultimately *subverted* to a chain of existing code snippets (called *gadgets*) for conducting some attack logic. ROP is Turning-complete on a wide variety of platforms [8, 24, 37], threatening the security of an overwhelming amount of systems and applications written in languages that inherently lack bounds checking, such as C/C++.

To thwart such an attack, existing software-based efforts mainly advocate two lines of defense mechanisms. One is to maintain the *control flow integrity* (CFI) by systematically verifying branch targets at runtime [25, 27, 28, 43, 46, 47]. This requires the recompilation of source code and could lead to *incompatibility* issues when securing code-on-the-shelf (COTS) binaries. Also, the trade-off between *performance* and *security* is tricky to navigate. In solutions where reasonable runtime overhead can be achieved, the enforcement of CFI is relatively weak and thus numerous attacks are feasible [9, 16–18].

Another direction is to proactively *invalidate* the prior knowledge that is necessary to launch an attack. To that end, a defender can *randomize* the memory layout in different granularity so that the gadget locations are not deterministic [23, 32, 33], or recompile her programs in a way that the exploitable gadgets are *eliminated* mostly [26, 29]. Again, for the latter case, recompilation can cause significant concerns on the portability, while for the former case, the system is still vulnerable to just-in-time (JIT) attacks [4, 38, 39] due to the static nature of the memory layout at runtime.

In this paper, we envisage the opportunity of devising a more desirable defense on the hardware technique of *context-sensitive decoding* (CSD) [42]. Basically, CSD works on the *decode* stages in the instruction pipeline, and it enables customizable *native-to-microop* translation in adherence to the dynamically changeable *context* such as hardware events or register updates. Leveraging the transparent nature of CSD as to attackers, we propose to customize the translation of `call/ret` instructions in a way that return addresses can be verified before being used, with copies that are previously stored within a well-protected region in virtual memory space, called a *shadow stack*.

While there already exists software ways to implement a similar idea [12, 31, 34], we aim to be distinct from them by *simultaneously* tackling three challenges. *First*, we need to safeguard the content in the shadow stack (*Security*). While it is infeasible for software-based methods to achieve this due to their agnostic to the lower-level system and hardware, we propose to perform strict *access control* on the shadow stack by only allowing a narrow range of micro-ops to modify the shadow stack (§ 5). *Second*, we need to accommodate exceptional scenarios where *non-local returns* are performed and the LIFO nature of shadow stack becomes invalid (*Functionality*). To that end, apart from bookkeeping return addresses, we advocate jointly storing the associated *stack pointers* for differentiating a software exception and an actual attack (§ 6). *Last but not least*, a practical defense is desired to exhibit near-to-optimal runtime overhead. To satisfy this requirement, on decoding return instructions, we *delay* the access to the shadow stack and refer to *return address stack* (RAS) first, a high-speed on-chip unit that possesses similar behaviors as the shadow stack (§ 7).

*Work done as an intern at University of California, San Diego.

The whole proposal is termed CSDrop and is integrated into the gem5 simulator. To showcase its effectiveness in thwarting ROP, we craft multiple ROP instances and launch them with the aid of the RIPE attack suite [44]. We compare the attack outcomes with and without CSDrop and the results prove that CSDrop fulfills the security requirement. We have also demonstrated the correctness of how CSDrop deals with exceptional semantics brought by `setjmp/longjmp` function calls. We finally test the runtime overhead of CSDrop with 16 CPU-intensive benchmark programs from SPEC CPU 2006 [22]. Results show that when RAS is utilized, the average overhead that CSDrop brings to the execution of protected programs is negligible (0.76% on average). To our knowledge, CSDrop is the *first* implementation of shadow stacks that tackle the three aforementioned challenges in the meantime.

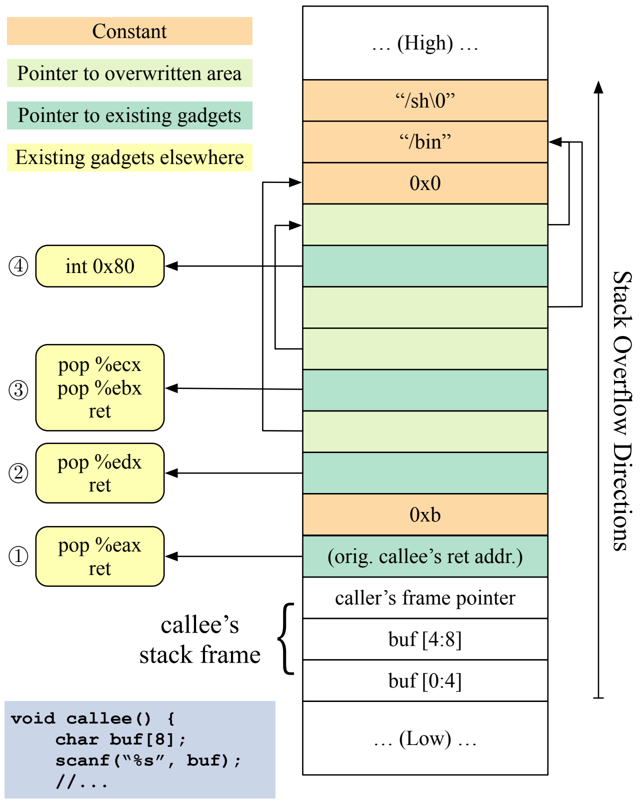
2 Background and Motivation

2.1 Return-Oriented Programming

A *buffer overflow* is a software bug that occurs when data that are stored adjacent to the end of an allocated array are corrupted due to a lack of bounds checking. Gaining notoriety since 1988 [40], buffer overflow vulnerabilities have long been one of the major security loopholes that plague the Internet, and it still ranks *second* in China National Vulnerability Database [2] today, just behind cross site scripting. These vulnerabilities have been systematically exploited by code reuse attacks, wherein *return-oriented programming* (ROP) [37] has been one of the most general and flexible schemes. A ROP attack can subvert control flow with the aid of *existing code gadgets* in memory image, instead of any injected code. Figure 1 exemplifies how a ROP attack that mounts on x86 executables can spawn a command shell.

The demo attack begins with injecting an carefully crafted payload (shaded area on the stack), which leverages buffer overflow when providing input to the `buf` array in `callee()`. In particular, the return address of `callee()` is overwritten with the address of a short code snippet ① in the program that ends with an return instruction (`ret`), called *gadget* by convention. Recall that what the CPU does when the instruction pointer reaches a return instruction is to (1) pop what the stack pointer currently refers to and (2) have the instruction pointer updated to that address. Upon executing the return instruction of `callee()`, instead of returning to its caller function as expected, the control flow is actually hijacked to gadget ①. Once the gadget has executed, the value of the `eax` register will be set to `0xb`, and the exploits continue with gadgets ②, ③, and ④. As a consequence, the necesary context of trapping into kernel and opening a command shell progressively get prepared, with which the attack can evolve into a more general and powerful one.

Note that the gadgets that ROP utilizes are not intentionally placed by victim programmers. Instead, they are so *short* in size and *common* in function that they can be *widely* found in *general-purpose libraries* of all kinds that most of C/C++ programs and drivers have to lean on, such as `libc`. It has been shown by extensive studies how to easily identify a *Turing complete* set of gadgets on different instruction set architectures including but not limited to x86 [37], ARM [24] and SPARC [8]. We also showcase the feasibility of conducting the above demo attack in our experimental environment later in Section 9.2. It is also noteworthy that so far, several evasive variants of ROP have been proposed such as Jump-Oriented Programming (JOP) [5, 11], Call-precede ROP [10], and Sigreturn-Oriented Programming (SROP) [7], whereas in this paper we focus on the defense against ROP.



```
void callee() {
    char buf[8];
    scanf("%s", buf);
    //...
```

Figure 1. Possible form of return-oriented programming.

2.2 Limitations of Software Defense

Existing software attempts to defend against ROP can be broadly classified into two categories of works.

Integrity Protection Intuitively, ROP can be thwarted by enforcing the integrity of return addresses stored in the stack. Early defenses have explored such a possibility by proposing *shadow stacks*, where software is re-compiled to additionally store copies of return addresses in a protected memory area at runtime for validation use [12, 31, 34]. To provide wider protection, Abadi, et al. first formalized the idea of *control flow integrity* (CFI), where the execution of a program is constrained in a predefined control flow graph by systematically checking target addresses of branches, jumps, calls, and returns [27]. Since then, there have been extensive follow-up works on CFI at different levels of granularity [25, 28, 43, 46, 47]. A practical issue of this line of works is the *tricky trade-off* between performance degradation and security guarantee. Implementing CFI statically can reap the most benefits of online overhead but leave the attack surface the largest due to the ignorance of the runtime dynamics, whilst enforcing CPI dynamically behaves in the opposite way [15]. Moreover, in the literature several *backdoor attacks* [9, 16–18] are shown to be possible to bypass these techniques, calling for a stricter enforcement of CFI.

Knowledge Invalidation The other direction of defending against ROP takes effect in a more aggressive manner by invalidating the *a priori* knowledge that a successful attack relies on. A major approach in this scope is *address space layout randomization*, whereby the memory layout of a program is stochastically determined to harden the utilization of gadgets. This can work in different levels of granularity including memory segment [33], code block [23], as well as instruction [32], with external help from the operating system and/or compilers. Due to the limited randomization entropy and intrinsic static nature, however, these works cannot provide foolproof security and are vulnerable to brute-force or just-in-time (JIT) attacks [4, 38, 39]. Apart from perturbing the locations of gadgets, another trial in this line of works advocates proactively *eliminating* gadgets upon program compilation [26, 29]. Such a method demands the presence of the source code of both a program and its dependent libraries, which makes it less appealing in practice, especially when only code-on-the-shelf (COTS) binaries are available.

Summary Table 1 summarizes the limitations of the above mentioned strategies, which implies that software-only solutions may not be the optimal. This fact drives us to explore the possibility of *hardware* solutions, which are not only orthogonal to the previous software-based attempts, but also has the potential to strike a new balance between the listed dimensions. In this paper, we aim to devise one of such possibilites with the following *requirements* born in mind:

1. *Security* (**R1**): effective in thwarting ROP.
2. *Functionality* (**R2**): letting the code work as usual.
3. *Performance* (**R3**): close-to-optimal runtime overhead.
4. *Portability* (**R4**): feasible for unmodified code.

3 Assumptions and Threat Model

Existence of Buffer Overflow We do not need programmers to change their habits or reframe their applications such that buffer overflow bugs are eliminated. Instead, a buffer overflow can still occur.

Complete Disclosure We assume that the attacker has full knowledge of our defense mechanism. We also assume that the attacker has unfettered access to the binary, source code, and the program in execution, meaning that she has a complete list of all potential ROP gadgets in the memory image, with which she is capable of launching a ROP attack if no security is enforced.

Table 1. The goal of our hardware solution and the status of the counterparts, where ●, ◓, ○ represent best to least adherence to the property in the row, respectively.

	CFI	ASLR	Gadget elimination	Our goal
Security	◓	○	●	●
Portability	◓	●	○	●
Performance	◓	●	●	●

No Complementary Method We require no additional software-based techniques to defeat ROP, including CPI, ASLR, and attempts that try to reduce usable gadgets in a program as mentioned above, since CSDrop is orthogonal to most existing software mechanisms.

Hardware Integrity The attacker may have access to our instrumented CPU, cache and memory, and may be able to probe any data or access patterns through side channels of all kinds. However, we do not tolerate the destruction or modification that is performed on the hardware.

4 CSDrop Overview

CSDrop disables return-oriented programming by customizing the behaviors of return instructions. In this section, we provide an overview of how this can be achieved in modern CPUs. Without loss of generality, we base our design on the x86 in what follows.

4.1 Context-Sensitive Decoding

For simplifying CPU design and improve instruction-level parallelism, CISC ISAs such as x86 and ARM typically translate native instructions, called *macro-ops*, into their respective RISC-like instuctions, called *micro-ops* (μ-ops) [21, 36]. For example, as mentioned in Section 2.1, the return instruction (`ret`) may correspond to three μ-ops: `ld t1, %esp; addi %esp, %esp, dsz; weip t1` [1]. The set of these μ-ops is statically predefined in μ-ops ROM on chip, and it is the decode stage of the processor pipeline that takes charge of the translation at runtime [19].

To harness the untapped potential of the decoder, *Context-Sensitive Decoding* (CSD) has been proposed to enable customization of the micro-op translation at the microsecond fine granularity [42]. Figure 2 illustrates the major idea of CSD. When CSD is turned on, the fetched instruction (①) will be redirected to the custom decoder, which decodes it not only based on the micro-op program counter (μPC), but

[1]What they essentially mean is to pop the content pointed by `%esp` to `%eip` in three steps.

also the current *context* (`ctx`), i.e., the index of rules that specify how to perform the native-to-microcode translation (②). Finally, besides PC, μPC is also updated upon completion of decoding according to branch prediction results to accommodate speculative execution (③).

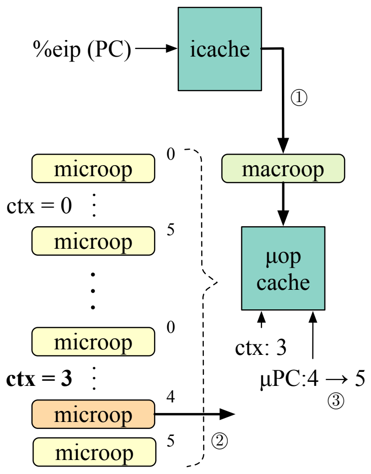


Figure 2. Context-Sensitive Decoding.

It is worth mentioning that such a technique does not come with significant hardware modification. The customization of micro-ops can be achieved by the exploitation of the well-established microcode update procedure outlined by Intel [21]. As for the context switch, it can be triggered by changes on the model-specific registers (MSRs). This means that programmers can switch on a particular context by requesting the operating system to properly set MSRs.

4.2 CSDrop Overview

At its core, CSDrop is an instantiation of CSD that allows return address validation for defending against ROP. Essentially, by customizing the decoding logic, we make the processor additionally store a copy of the return address upon the execution of a function call in a protected area, named *shadow stack*, and also make it load and validate the return address on the stack using the saved copy upon returning from that function. Figure 4 provides a sketch of the high-level idea of CSDrop. ① When the instruction counter of the current function `caller` reaches the place where function (`callee`) is called via `call`, additional micro-ops are added upon the decode stage to store the return address in the shadow stack. ② After that, the `call`'s original sequence of micro-ops is executed, which stores the return address in the user stack and jumps to the beginning of `callee`. ③ When it comes to the exit from `callee`, again, auxiliary micro-ops are added to verify the equivalence between the return address stored in the user stack and the one in the shadow stack. ④ It is only when verification passes can the control flow return to `caller`.

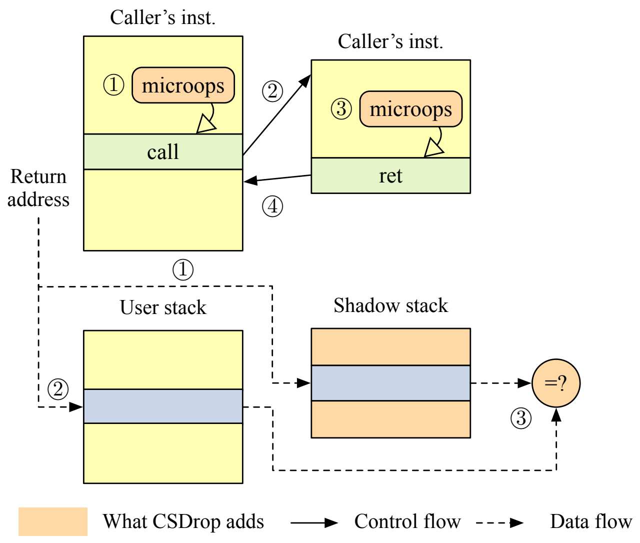


Figure 3. Idea sketch of CSDrop.

Note that as we have mentioned in Section 2.2, the idea of shadow stack has long been explored in the literature of software-based mechanisms. They implement the idea basically by inserting shadow-stack-related instructions into the source code or binary of an application. However, these methods failed to satisfy the aforementioned **R1**–**R4** *simultaneously*.

1. *Security* (**R1**): Working only on the application code, the trusted computing base (TCB) of software methods is large. More precisely, they can by no means ensure the integrity of their shadow stacks against other corrupted applications or the operating system. With sutble hardware change, we show how we manage to perform foolproof access control on the shadow stack in Section 5.
2. *Functionality* (**R2**): Few software-based proposals study the handling solution to exceptional instruction sequences where call and return instructions do not match, such as the ones generated by `setjmp/longjmp` from the standard C library. We provide our amending solution on such corner cases in Section 6.
3. *Performance* (**R3**): While software-based shadow stack solutions introduce extra instructions and thus induce sub-optimal performance cost, our hardware-based solution induces computation overhead more mildly. Moreover, we manage to further optimize the runtime overhead with the aid of the on-chip return address stack. We elaborate more on our performance optimization strategy in Section 7.
4. *Portability* (**R4**): Some of the software-based methods demand the availability of source code for successful instrumentation, while our hardware-based method is naturally agnostic to the code running above.

5 Access Control of the Shadow Stack

The security of CSDrop is rooted in the integrity of the shadow stack. In this section, we elaborate on details of its access control of the shadow stack in response to **R1**.

Basic Configurations While the shadow stack can be implemented in either the physical memory or another specialized one, we opt to the use of *physical memory* for minimizing the need for modifying the underlying hardware (**R4**), as well as avoiding the potential runtime overhead induced by re/storing the shadow stack upon context switch[2] (**R3**). Specifically, when a process starts, apart from the user stack, the operating system also maps a shadow stack to its virtual memory space. The shadow stack is designed to have the same length as the user stack does to avoid overflow[3], i.e., 8MB on the x86-64. Note that we do not allocate such a large chunk of memory immediately. Like the user stack, the occupation of the shadow stack is incremented page by page on-demand in physical memory.

Protection by TLB Similar to the access to the user stack, we manage the shadow stack via its own *stack pointer*, which records the top position in the stack. We also maintain a *frame pointer* for the shadow stack which points to the bottom for emptiness detection (this is necessary for the exceptional scenarios mentioned in Section 6). Both of the two pointers are *transparent* to any programmer for two reasons: (1) the reference to them merely exists in the level of micro-ops which a programmer cannot directly devise; (2) as other logical registers, they are exclusively mapped to some physical registers by the processor.

Given the safeguarded stack pointer and frame pointer, we can identify the up-to-date range of the shadow stack in virtual memory space. To control the access to the whole range, we make a subtle modification on the *translation lookaside buffer* (TLB), which now assists in address validation with the knowledge of shadow stack range. More precisely, only the micro-ops generated from `call`/`ret` under the intended context can refer to that range of space. Other micro-ops, e.g., the ones generated from a malicious `mov`, will be rejected on reading from or writing to that range with a hardware exception thrown to alert the application user. Note that there are still ways to probe the content within the range of shadow stack like performing cache-based side-channel attacks [20, 30, 45]. However, ROP attackers can still not able to bypass CSDrop as long as they are not permitted to *modify* the content in adherence to their choice. To conclude, the integrity of the shadow stack can be guaranteed in CSDrop.

[2]Otherwise, when the process switches to execute another process, the shadow stack of the current process in the specialized memory, if not sufficiently large, has to be securely stored in the main memory, e.g., with encryption involved, which is foreseeably time-consuming.

[3]In this case, the user stack will be exhausted earlier than the shadow stack be, as it contains not only return addresses but also many other context information and program data.

6 Handling Exceptional Semantics

To ensure the robustness of CSDrop as required by **R2**, we first ask the question:

- Is a *mismatch* between the return address in the user stack and that in the shadow stack a necessity and sufficiency of identifying a ROP attack?

According to our observation, the answer is negative and the mismatch is only a *necessary* condition for confirming ROP. The reason for this lies in an exceptional scenario where a function may *not* return to its caller at all. In this case, a mismatch can be triggered when the shadow stack is in use, though, it is not related to a launched ROP attack.

setjmp/longjmp In C, such a special scenario is mostly entered when a program or its dependent libraries calls `setjmp/longjmp` functions. Basically, `setjmp()` *stores* the context information of the current execution point to a buffer, while `longjmp()` causes that environment to be *restored*. This allows a program to quickly return to a previous location, *without* going through the chain of return addresses in between[4] [11, 44]. To further illustrate, we give a minimum example of the use of `setjmp/longjmp` as well as how a mismatch situation is resulted in Figure 4, along with Figure 5. We also simulate the execution step by step as follows.

1. `first()` calls `setjmp()` which saves the current context, including all the values in general-purpose registers and the return addresses in `first()`, into an predefined global buffer `buf` (Line 19). Figure 5a shows the status of both the user stack and the shadow stack before `setjmp()` returns. As for the return value, by design, if `setjmp()` is called the first time, it is 0, and thus `second()` will be called (Line 21).
2. As `second()` and `third()` are executed in turn, it ends up with the calling of `longjmp()` (Line 7). What the `longjmp()` does include: (1) reload the general-purpose registers with the corresponding values stored in `buf`; (2) change the return value (which is stored in a specific register like %rax in the x86-64) to a non-zero value[5] (1 in this example); and (3) use unconditional branch instruction `jmp` to return to the address that has been previously stored in `buf`. As a result, it will look as if the program returns from `setjmp()` to `first()` as it previously did with all the context remains the same, except for the return value. The status of the two stacks before and after `longjmp()` returns are depicted in Figure 5b and Figure 5c, respectively.

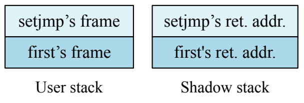
As Figure 5c shows, after returning from `longjmp()`, the sequence of return addresses stored in the shadow stack will become inconsistent with that in the user stack. Should it be recognized as an appearance of ROP, the innocent control

[4]This is basically why `setjmp/longjmp` are named so.

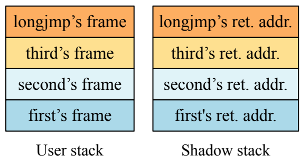
[5]As a result, after returning to `first()`, the processor can carry on executing `auxiliary()`, instead of trapping in the dead loop of `first–second–third–longjmp–first`.

```c
#include <stdio.h>
#include <setjmp.h>
static jmp_buf buf;

void third() {
    printf("third\n");
    longjmp(buf, 1);
    printf("impossible to get here\n");
}

void second() {
    printf("second\n");
    third();
    printf("impossible to get here\n");
}

void first() {
    printf("first\n");
    if (!setjmp(buf)) {
        printf("if\n");
        second();
    } else
        printf("else\n");
    auxiliary();
}

void main() {
    printf("main\n");
    first();
    printf("back to main\n");
}
```

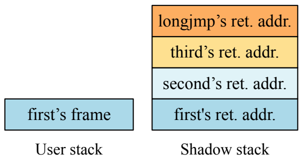
Figure 4. A toy code example that uses `setjmp/longjmp`, where `auxiliary()` can be arbitrarily complex and we thus omit its definition.



(a) Right after calling `setjmp`.



(b) Right after calling `longjmp`.



(c) After `longjmp` jumps.

Figure 5. Snapshots of the two stacks during the execution of the code in Figure 4.

flow will break, which is not desirable. Note that it is not uncommon to come across with `setjmp/longjmp` pairs even if the program to run does not explicitly contain them, as they are necessities to implement exception handling in C[6] and thus are widely used in the operating system and many general-purpose libraries [41].

Vulnerabilities in Vanilla Repetitive Checking An intuitive solution to differentiating the above exceptional semantics and a real ROP attack is to keep popping the shadow stack and repeat address comparison whenever a mismatch is detected [12], and it is only when there is no match found upon the exhaustion of the shadow stack that we confirm the existence of a ROP attack. We name such a scheme *vanilla repetitive checking* in this paper. The rationale behind vanilla repetitive checking is that the root cause for the troublesome semantics brought by `setjmp/longjmp` is their *agnostic* to the shadow stack. Thus, having the shadow stack *simulate* the behavior of the user stack should help. For example, after sequentially popping the return address of `longjmp`, `third` and `second`, we can then find a match in the case of Figure 5c, from which we are safe to carry on execution as this could not happen in case of a ROP attack.

However, vanilla repetitive checking has a security *loophole* under our threat model (§ 3). Recall that an adversary in our settings is able to *corrupt* the return addresses stored by `setjmp`, causing the control flow subverted to anywhere she wants after the execution of the associated `longjmp`. Such corruption is actually a one-gadget ROP. The instructions between the hijack destination instruction and its nearest return instruction actually form a ROP gadget. Although the gadget may not be able to perform a targeted attack due to the lack of cooperation with other gadgets, still, it suffices to break the normal execution of a program, which raises non-trivial security concerns. The practical issue of vanilla repetitive checking is that it is not able to detect such type of ROP, *as long as* the gadget is located in functions that are not returned.

More concretely, consider the example in Figure 4 and Figure 5, again. If the return address stored by `setjmp()` is modified to an address that points to somewhere in the middle of `auxiliary()`, then the execution of `longjmp()` may cause the control flow to start from that place, i.e., `auxiliary()` will only be executed *partly*, instead of completely. Note that such abnormal behavior is transparent to vanilla repetitive checking, as it only cares about whether the return address stored at the top of the shadow stack matches one stored in the current stack frame when `first()` returns. In a word, vanilla repetitive checking does not close the backdoor for the special type of ROP mentioned above.

Enhanced Repetitive Checking To make it harder to bypass repetitive checking, we enhance the defense by also taking into account the integrity of the *stack pointer*. More precisely, when making a call to a function, we store both the

[6]The call to `setjmp` is analogous to a `try` statement in other languages, while the call to `longjmp` is like `throw`.

return address and the current stack pointer in the shadow stack. Accordingly, when the function returns, we not only verify the return address, but also the stack pointer. The *intuition* here is that, according to our observation, the stack pointer is highly *versatile*. This means that if a function is executed with different portions, the stack pointer is likely to end up with different locations. In other words, if a function does not return to its caller at the expected return address but some point later, we can identify such misbehavior with significant probability at the end of the caller function by verifying the stack pointer. It is worth mentioning that while we are not the only one to advocate jointly verifying return addresses and stack pointers [13, 31], we are the *first* to discuss the security benefit of doing so.

7 Performance Optimization

In adherence to **R3**, we propose to optimize the runtime overhead of shadow-stack-related operations with the aid of the return address stack which already exists in a processor.

Return Address Stack High-performance processors typically predict the next fetch address in parallel to the instruction cache access for improving the flow in the instruction pipeline. In particular, the prediction of the target of return instructions is treated as a special case and is performed by a specialized hardware unit called *return address stack* (RAS) [19]. The RAS is a last-come-first-out (LIFO) structure, where every time the processor makes a function call, the address of the next instruction is regarded as the return target and thus pushed in. When a return instruction is fetched[7], the youngest entry in RAS is then popped out and used as the predicted destination. In this sense, RAS resembles the shadow stack in CSDrop much, both in the working behavior and the transparency to programmers. Given the *on-chip* nature of RAS, the performance of managing the shadow stack can be significantly boosted if the stack is implemented in RAS, instead of on caches and the physical memory[8].

However, RAS cannot be directly used as the shadow stack in CSDrop for its failure of fulfilling **R2** in two aspects. *First*, originally introduced for accelerating return target prediction, RAS only accounts for the prediction for normal call/ret pairs. It cannot accommodate exceptional scenarios as mentioned above (§ 6), neither can it be programmed to perform repetitive checking. *Second*, as an on-ship hardware unit, RAS is small in size (typically tens to hundreds of slots) and thus easy to overflow. In case of a RAS overflow, even if there is no ROP attack presenting, a mismatch between the return address in the current stack frame and the top entry in RAS can still occur due to the loss of address copies. Note that the mess brought by RAS overflows cannot be tolerated even if they seldom happen, as we aim to ensure the functionality of protected programs in any case.

Leveraging the Assistance from RAS Still, there is a way to utilize the fast access nature of RAS to optimize the performance of CSDrop–to treat RAS as a *cache* of the shadow stack. To be exact, we *disable* the customized micro-op translation for any return instruction in the first place and execute it as usual. Depending on the return target predicted by RAS, we take further actions accordingly:

- *RAS hit*: If the predicted target matches the return address popped from the current stack frame, it implies that no ROP attack has been mounted. In this case, we do not need to bother accessing the shadow stack. Instead, we only have to *conceptually* pop, i.e., through pointer decrement without actual access, the corresponding return address and stack pointer stored in the shadow stack for maintaining the consistency between the shadow stack and the user stack.
- *RAS miss*: If a mismatch occurs, it may or may not result from a RAS overflow. We thus *retire* the return instruction and *decode* it again with customized micro-ops to make use of the shadow stack.

Since in most cases the depth of function calls does not exceed the capacity of RAS, we can frequently get rid of the cost of accessing the shadow stack. In Section 9, we will empirically evaluate the performance bonus brought by such an optimization strategy.

8 Implementation

In brief, we have integrated CSDrop into the `gem5` simulator [3] in evaluated in `system-call emulation` (SE) mode, with around 1500 lines of code in total. The CPU we base on is `DerivO3CPU` on the `x86-64`, which is consistent with most of the modern processors with advanced features realized including pipelining, speculative execution as well as out-of-order execution.

8.1 Integration in gem5

In this section, we elaborate on necessary details of the integration efforts in `gem5`. While we will also provide the detailed code in the supplementary materials, readers can refer to Figure 6 for gaining a logical view on which parts of the computer architecture have been engineered, or at least involved, in the first place.

The Shadow Stack In the virtual memory space of the x86-64, bytes indexed from `0x00000000_00000000` to `0x00007FFF_FFFFFFFF` composes user space, while bytes indexed from `0xFFFF8000_00000000` to `0xFFFFFFFF_FFFFFFFF` comprises the kernel space. Thus, we allocate a shadow stack up to 8MB for each process in between the two parts to avoid bothering with the kernel as well as preventing any possible conflict with users' applications (`src/sim/process.hh` and `process.cc`). We identify the current range of the shadow stack by

[7] Or, is predicted to be fetched by the branch target buffer (BTB).

[8] Even if the whole shadow stack is cached near to the processor, e.g., in L1 cache, the access to RAS still looks faster as it takes place in the decode stage, while the access to the shadow stack happens in the execution stage.

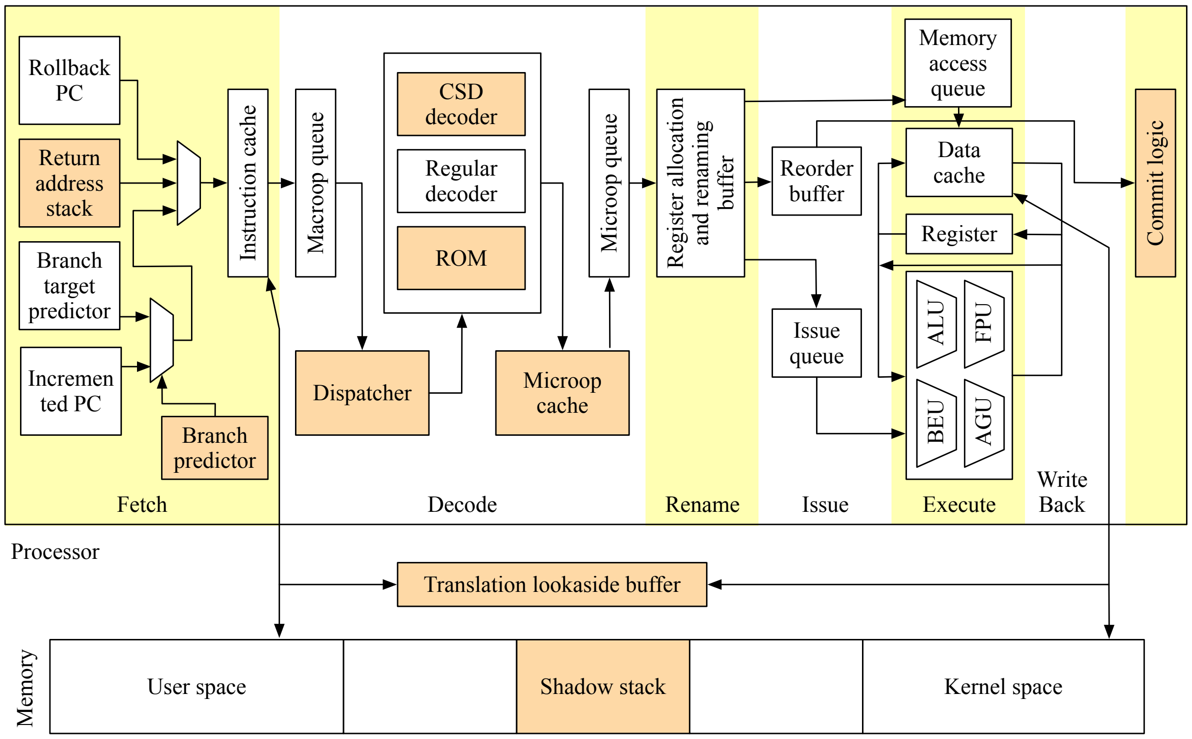


Figure 6. Modifications (orange shaped part) that CSDrop makes on `gem5`.

using two logical registers, `t10` and `t11`, which record the top and the bottom, respectively. We also define a previlige for accessing the shadow stack `src/mem/request.hh`, and grant that previlige to no microops but the CSDrop-related ones (`src/cpu/base_dyn_inst.hh`). With that, the TLB is able to perform access control (`src/x86/tlb.cc`) on the shadow stack for every data access microop, as detailed in Section 5.

Microop Customization As mentioned in Section 4.1, the native-to-microop translation in context sensitive decoding (CSD) depends on both the context information (`ctx`) and the current microop program counter (`microPC`). With CSD enabled, when decoding an instruction all of its associated microops are first gathered (`src/arch/x86/decoder.cc`). Then given the `ctx` and `microPC` that are determined in the fetch stage (`src/cpu/o3/fetch_impl.hh`), the decoder is able to pick up the proper microop to execute (`src/arch/x86/insts/macroop.hh`). Atop the CSD framework, we customize the translation rules, i.e., the mapping of instruction/context/microPC to microop, of call and ret instructions (`src/arch/x86/isa/insts/general_purpose/control_transfer/call.py` and `xreturn.py`), to realize the design described in Section 6.

Interaction with RAS As elaborated in Section 7, we perform normal decoding for return instructions and only re-encode to involve the shadow stack when a RAS miss occurs. In practice, RAS misses can be observed when the return target is computed at the execute stage (`src/cpu/o3/iew_impl.hh`). This result can then be shared with other stages in the pipeline via different `TimeBuffers`. Upon a RAS miss, the commit stage aborts the related speculatively executed instructions, while the fetch stage updates `ctx` and `microPC` for re-encoded the corresponding return instruction.

Dynamic Instructions In `gem5`, the data structure for an instruction is static, meaning that originally, instructions of the same address are tightly coupled in terms of `ctx` and `microPC` when they appear concurrently in the pipeline, which causes unintended behaviors as they are expected to be independent. We sidestep this pitfall by adding additional attributes to that data structure so that instruction instances are dynamic, i.e., they can be uniquely distinguished (`src/arch/x86/insts/macroop.hh`). This attributes to most of the complexity and debugging efforts of CSDrop, however, we defer the details to the code.

System Call Binding In `syscall emulation (se)` mode, system calls cannot be directly emulated, which hardens the demonstration of how ROP can subvert the control flow in the absence of CSDrop. To circumvent, we bind system calls in the simulator to those in the host machine (`src/sim/syscall_emul.hh`) such that a ROP launched in an application running atop the simulator can make system calls that take effect on the host machine.

8.2 Auxiliary Tools

To showcade the defense effectiveness of CSDrop, the testing application running atop our modified gem5 should attempt to launch a ROP attack. To that end, we resort to two additional useful tools.

The RIPE Attack Suite [44] is an attack testbed proposed to quantify the protection coverage of any defense measure by performing a wide range of buffer-overflow attacks. More concretely, the software itself is both an attacker and a victim. It is programmed to expose a buffer overflow vulnerability, and at runtime, to attack itself by loading a predefined malicious payload to the associated buffer. To perform ROP attacks, we need to prepare a payload that overwrites the user stack with necessary gadget pointers and immediate data, similar to what is exemplified in Figure 1. A successful construction of the payload requires the runtime knowledge of the stack layout, as well as the addresses of exploitable gadgets in the software or its included libraries.

ROPGadget [35] is a powerful tool that facilitates the searching process of ROP gadgets on a wide variety of platforms such as x86, ARM, and MIPS. For statically linked binaries, the addresses ROPGadget calculates are the same as what will be at runtime. For dynamically linked libraries, on the other hand, what are reported by ROPGadget are relative to the code segment, which requires that the absolute address of the code segment be determined *a priori*.

9 Evaluation

We evaluate the effectiveness of CSDrop under realistic settings. We organize our evaluation by assessment angle with the following key results.

- CSDrop can thwart multiple ROP instances, fulfilling the security requirement **R1**.
- CSDrop is shown to properly handle the expectional semantics induced by `setjmp/longjmp` pairs, satisfying the functionality requirement **R2**.
- The fully optimized CSDrop can induce an average overhead as subtle as 0.76%, meeting the performance requirement **R3**.

9.1 Methodology

Experimental setup All the experiments are conducted on a Ubuntu 16.04 LTS host machine with 4.4.0 as the Linux kernel version and x86-64 as the underlying instruction set architecture. The host machine is equipped with a 8-core Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz CPU and 32GB Memory. The used gem5 simulator is of version `stable_2015_09_03` and it runs in `syscall emulation (se)` mode. During simulation, we enable cache, use `x86_detailed` as CPU type and 8GB as memory size. For each benchmark program in SPEC CPU 2006, we run till its termination or reaching an instruction count of 2×10^9.

Table 2. The gadgets in RIPE that we use in launching ROPs.

Starting Address	Gadget Content
0x491176	pop rax; pop rdx; pop rbx; retq
0x44e140	mov eax 0x3b; syscall
0x404e16	pop rdi; ret
0x4536e6	pop rdx; ret
0x404f37	pop rsi; ret
0x453709	pop rdx; pop rsi; ret
0x40044a	syscall

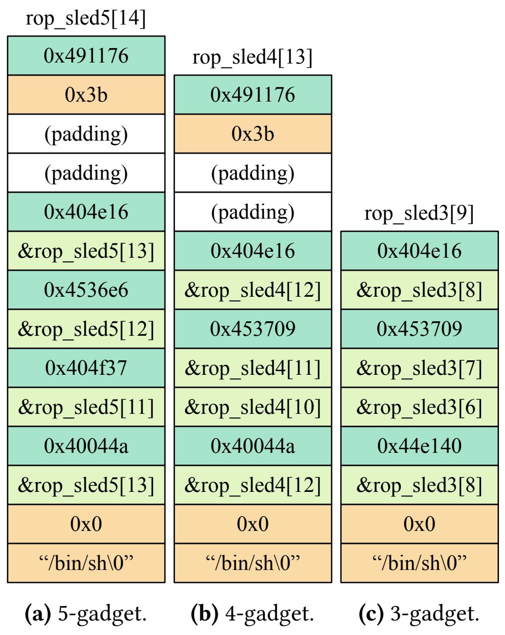
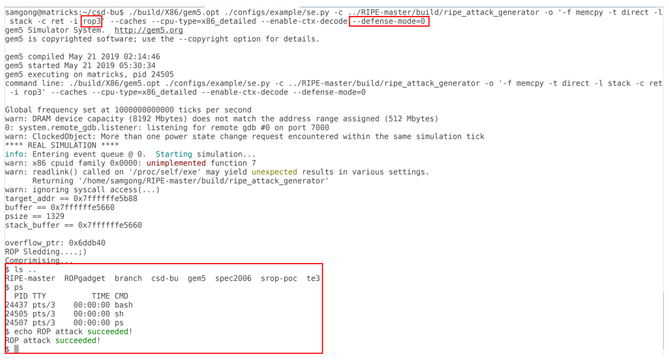


Figure 7. The buffer overflow payload for our ROP attacks with different numbers of involved gadgets.

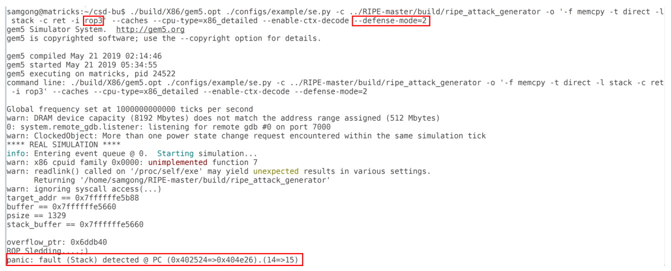
Datasets and Benchmark Programs As mentioned in Section 8.2, we utilize the RIPE attack suite to simulate ROP attacks. Specifically, we choose to launch ROP leveraging the buffer overflow vulnerability that is located on `stack` and exposed by `memcpy()`. As there is no standard benchmark for testing the handling of `setjmp/longjmp` semantics, we use the homebrew C program as previously given in Figure 4 and analyzed in Section 6. Finally, for measuring the performance impact atop gem5 simulator, we use benchmark programs from SPEC CPU 2006 [22], a CPU-intensive benchmark suite standardized by the industry.

Metrics We care about whether the testing programs return expected results (e.g., prompting expected character strings) when evaluating the security and functionality. When examining the performance, we concern the metrics `system.cpu.numCycles` indicated by gem5. We are also interested in the hit rate of the RAS, which can explain the performance



```
samgong@matricks:~/csd-bu$ ./build/X86/gem5.opt ./configs/example/se.py -c ../RIPE-master/build/ripe_attack_generator -o '-f memcpy -t direct -l
 stack -c ret -i rop3' --caches --cpu-type=x86_detailed --enable-ctx-decode --defense-mode=0
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled May 21 2019 02:14:46
gem5 started May 21 2019 05:30:34
gem5 executing on matricks, pid 24505
command line: ./build/X86/gem5.opt ./configs/example/se.py -c ../RIPE-master/build/ripe_attack_generator -o '-f memcpy -t direct -l stack -c ret
 -i rop3' --caches --cpu-type=x86_detailed --enable-ctx-decode --defense-mode=0

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb.listener: listening for remote gdb #0 on port 7000
warn: ClockedObject: More than one power state change request encountered within the same simulation tick
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
warn: x86 cpuid family 0x0000: unimplemented function 7
warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
      Returning '/home/samgong/RIPE-master/build/ripe_attack_generator'
warn: ignoring syscall access(...)
target_addr == 0x7ffffffe5b88
buffer == 0x7ffffffe5660
psize == 1329
stack_buffer == 0x7ffffffe5660

overflow_ptr: 0x6ddb40
ROP Sledding....;)
Comprimising...
$ ls ..
RIPE-master  ROPgadget  branch  csd-bu  gem5  spec2006  srop-poc  te3
$ ps
  PID TTY          TIME CMD
24437 pts/3    00:00:00 bash
24505 pts/3    00:00:00 sh
24507 pts/3    00:00:00 ps
$ echo ROP attack succeeded!
ROP attack succeeded!
$ 
```

(a) Without CSDrop.



```
samgong@matricks:~/csd-bu$ ./build/X86/gem5.opt ./configs/example/se.py -c ../RIPE-master/build/ripe_attack_generator -o '-f memcpy -t direct -l
 stack -c ret -i rop3' --caches --cpu-type=x86_detailed --enable-ctx-decode --defense-mode=2
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled May 21 2019 02:14:46
gem5 started May 21 2019 05:34:55
gem5 executing on matricks, pid 24522
command line: ./build/X86/gem5.opt ./configs/example/se.py -c ../RIPE-master/build/ripe_attack_generator -o '-f memcpy -t direct -l stack -c ret
 -i rop3' --caches --cpu-type=x86_detailed --enable-ctx-decode --defense-mode=2

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb.listener: listening for remote gdb #0 on port 7000
warn: ClockedObject: More than one power state change request encountered within the same simulation tick
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
warn: x86 cpuid family 0x0000: unimplemented function 7
warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
      Returning '/home/samgong/RIPE-master/build/ripe_attack_generator'
warn: ignoring syscall access(...)
target_addr == 0x7ffffffe5b88
buffer == 0x7ffffffe5660
psize == 1329
stack_buffer == 0x7ffffffe5660

overflow_ptr: 0x6ddb40
ROP Sledding....;)
panic: fault (Stack) detected @ PC (0x402524=>0x404e26).(14=>15)
```

(b) With CSDrop.

Figure 8. The effectiveness of performing the 3-gadget ROP in the absence and presence of defense from CSDrop.

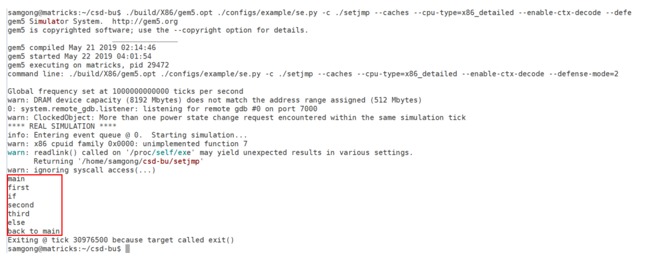
gain by leveraging RAS. This can be derived by collecting two additional metrics: `system.cpu.branchPred.usedRAS` and `system.cpu.branchPred.RASInCorrect`.

9.2 Defense Effectiveness

The foremost task of CSDrop is to defend against ROP, so we first verify its security under realistic attacks. We aim to construct three ROP attacks that rely on 3, 4, and 5 gadgets, respectively. For each attack instance, we first disassemble the binary of the original RIPE to obtain its x86 assembly code and the corresponding memory layout on load. We then use ROPGadget to search for prospective gadgets, from which we opt to use the ones as listed in Table 2. We then carefully craft the buffer overflow payload as detailed in Figure 7, with the purpose of opening a shell upon success. We then hardcode the payload in the source code of RIPE and finally recompile RIPE. In expectation, when the modified RIPE is run atop `gem5`, it should utilize the predefined buffer overflow vulnerability to attack itself. If CSDrop is disabled, nothing should prevent the control from being subverted to *host's shell program*. Otherwise, the attack should be detected and a corresponding *exception* should be thrown to notify the RIPE user. In reality, all the tests *pass*. For brevity, we here only report the screenshots of performing the 3-gadget ROP with and without defense backed by CSDrop in Figure 8. As depicted, when CSDrop is absent, our 3-gadget attack is successfully launched and the attacker can thereby execute shell command of her choices such as `ls`, `ps` and `echo`. On the other hand, if CSDrop is enforced, no shell is opened but a customized `panic` message of `gem5` is prompted instead. Note that our homebrew attacks are *representative* as their construction is strictly aligned with the standard method of launching a ROP. We hence conclude that CSDrop effectively thwarts ROP, fulfilling the security requirement **R1**.

9.3 Setjmp/longjmp Handling

We next delve into the effectiveness of CSDrop in preserving the functionality of the program it protects. Although



```
samgong@matricks:~/csd-bu$ ./build/X86/gem5.opt ./configs/example/se.py -c ./setjmp --caches --cpu-type=x86_detailed --enable-ctx-decode --defe
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled May 21 2019 02:14:46
gem5 started May 22 2019 04:01:54
gem5 executing on matricks, pid 29472
command line: ./build/X86/gem5.opt ./configs/example/se.py -c ./setjmp --caches --cpu-type=x86_detailed --enable-ctx-decode --defense-mode=2

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb.listener: listening for remote gdb #0 on port 7000
warn: ClockedObject: More than one power state change request encountered within the same simulation tick
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
warn: x86 cpuid family 0x0000: unimplemented function 7
warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
      Returning '/home/samgong/csd-bu/setjmp'
warn: ignoring syscall access(...)
main
first
if
second
third
else
back to main
Exiting @ tick 30976500 because target called exit()
samgong@matricks:~/csd-bu$ 
```

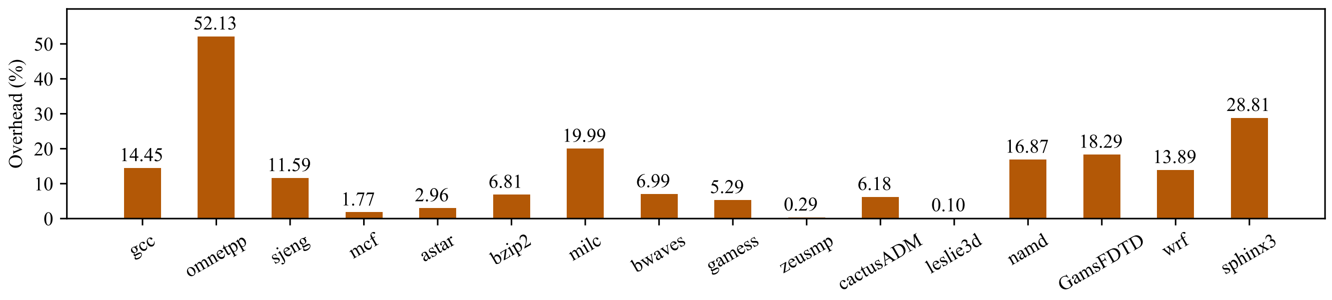
Figure 9. The execution result of the program described in Figure 4 atop CSDrop-enabled `gem5`.

the evaluation in Section 9.2 already implies such a property[9], in this section we explicitly and thoroughly test this aspect with intentionally introduced `setjmp/longjmp` calls. As mentioned in Section 6, CSDrop should be able to handle the mismatch of return addresses by correctly precluding the disruption brought by `setjmp/longjmp`. As for running the example program in Figure 4, the expected output to `stdout` should be:

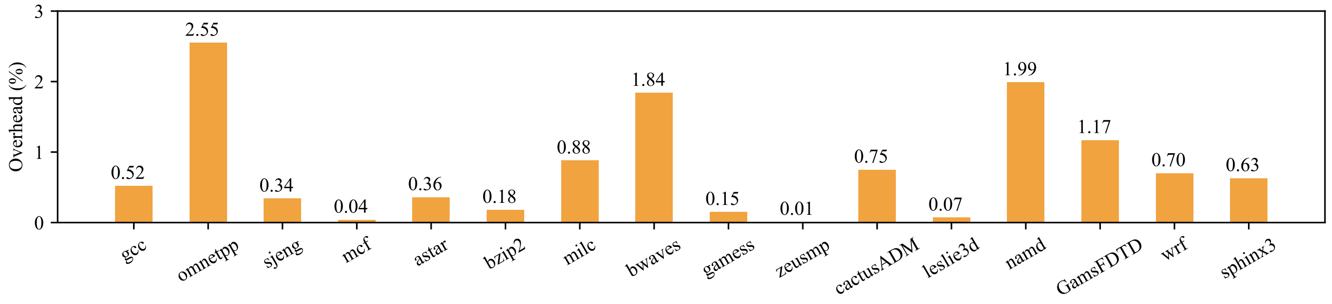
```
main
first
if
second
third
else
back to main
```

As shown in Figure 9, we get exactly the *expected* result when running this program atop CSDrop-enabled `gem5`, instead of emitting any false positive detection signal. Again, we do not need other testing programs as the used program *suffices* to prove the effectiveness of CSDrop on dealing with exceptional semantics induced by `setjmp/longjmp`. In short, CSDrop can satisfy the functionality requirement **R2**.

[9]Otherwise the modified RIPE will not attempt to perform the predefined attack and the exception will thus not be thrown, with significant probability.



(a) Without RAS.



(b) With RAS.

Figure 10. The runtime overhead of CSDrop with and without the utilization of RAS.

Table 3. RAS hit rates summary for CSDrop with the 16 SPEC CPU 2006 programs.

Program	RAS Hit Rate	Program	RAS Hit Rate	Program	RAS Hit Rate	Program	RAS Hit Rate
gcc	0.999938592	astar	0.99999983	gamess	0.99987879	namd	0.999985737
omnetpp	0.999603741	bzip2	0.99999824	zeusmp	0.999649639	GemsFDTD	0.99999432
sjeng	0.99998018	milc	0.99999775	cactusADM	0.99994351	wrf	0.999998083
mcf	0.99971583	bwaves	0.999998634	leslie3d	0.998435288	sphinx3	0.99999938

9.4 Runtime Performance

It finally comes to the evaluation of the impact that CSDrop has on the end-to-end performance of the original system. As we are conducting experiments on top of gem5, collecting the simulated running time of benchmark programs is meaningless. Instead, we resort to the number of *clock cycles*, which is proportional to the actual running time in reality. For each of our selected 16 benchmark programs that are drawn from SPEC CPU 2006, we run *three* passes of experiments: the first is with unmodified gem5, the second and the third one are with CSD-assisted gem5 with and without RAS as a means of performance optimization, respectively. By comparing the results of the first two passes, we obtain the results as illustrated in Figure 10a, while by comparing the results of the first and last pass we attain the results as shown in Figure 10b.

We first notice that even without the involvement of RAS, the performance overhead of CSDrop is *mild*, with the mean being 12.90% and median being 9.29%. The maximum overhead, which occurs in the execution of omnetpp, is only 52.13%. This is not only acceptable but outperforms most of the software implementations of shadow stack including ROPdefender [14] whose average overhead on integer benchmark programs and floating-point ones are 41% and 117%, respectively. Moreover, the performance potential of CSDrop is *fully unlocked* when the hardware RAS gets involved: the average and median overhead is as low as 0.76% and 0.56%, respectively. This beats the state-of-the-art hardware shadow stack [13] whose average overhead is around 4.6%.

We further look into the *source* of such performance benefit by studying the RAS hit rate. As mentioned above (7), if the predicted target matches the return address that is popped from the stack, we can assure that there is no ROP launched and can thus be free from bothering with the shadow stack access. As summarized in Table 3, the RAS hit rate is high enough, with the mean and median being 99.99% and 99.99%, respectively. This explains why we can avoid accessing our shadow stacks most of the time and thus reap the maximum performance benefits by realizing a hardware shadow stack. To conclude, CSDrop exhibits near-optimal runtime cost, meeting the performance requirement **R3**.

10 Discussion

Portability CSDrop is a general solution that can be widely adopted in reality for the following three reasons:

1. It is orthogonal to other defense mechanisms and thus does not rely on the presence of them, as mentioned in our threat model (§ 3).
2. It operates at the CPU level and thus does not require any modification or recompilation of applications.
3. The modification that CSD and CSDrop perform on the CPU is feasible with the well-established microcode update procedure, as mentioned in Section 4.1.
4. Although we demonstrate the feasibility atop the x86-64 architecture, CSDrop does not rely on any specialized hardware unit. In other words, we expect it can be seamlessly ported to other platforms such as ARM.

11 Related Work

Hardware Shadow Stack As mentioned in Section 2.2, shadow stacks have long been explored in fighting against ROP attacks. However, most of the existing implementation are based on *software* design [12, 14, 31, 34], which is hard to realize the four requirements listed in Section 4.2. The work that is most closely related to ours is the parallel shadow stack that is implemented with the assistance of hardware [13]. As shown in Section 9.4, its performance overhead is one order-of-magnitude larger than that of CSDrop. Moreover, it does not explicitly describe their handling of the exceptional scenarios and potential security pitfall if dealt with wrongly as we do in Section 6. It is also noteworthy that Intel also includes hardware shadow stack in its Control-Flow Enforcement Technology (CET) [1]. However, there is no open-sourced documentation on its implementation details due to the interests of the business. We are thus curious about the differences between CSDrop and that design.

12 Conclusion

In this paper, we present CSDrop to secure applications from being attacked by ROP. As its core, CSDrop introduces subtle and feasible changes in the CPU to realize a hardware version of shadow stacks. Compared to related works, CSDrop is one of the few that achieves full security, untapped functionality, ideal performance, and wide portability at the same time.

References

[1] 2016. *Intel Releases New Technology Specifications to Protect Against ROP attacks.* https://software.intel.com/content/www/us/en/develop/blogs/intel-release-new-technology-specifications-protect-rop-attacks.html

[2] 2021. *China National Vulnerability Database of Information Security.* http://www.cnnvd.org.cn/

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[4] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. 2014. Hacking blind. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 227–242.

[5] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* 30–40.

[6] Harold Booth, Doug Rike, and Gregory A Witte. 2013. The national vulnerability database (nvd): Overview. (2013).

[7] Erik Bosman and Herbert Bos. 2014. Framing signals-a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 243–258.

[8] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security.* 27–38.

[9] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *24th {USENIX} Security Symposium ({USENIX} Security 15).* 161–176.

[10] Nicholas Carlini and David Wagner. 2014. {ROP} is Still Dangerous: Breaking Modern Defenses. In *23rd {USENIX} Security Symposium ({USENIX} Security 14).* 385–399.

[11] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security.* 559–572.

[12] Tzi-cker Chiueh and Fu-Hau Hsu. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems.* IEEE, 409–417.

[13] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security.* 555–566.

[14] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* 40–51.

[15] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation.* 75–88.

[16] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the point (er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy.* IEEE, 781–796.

[17] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* 901–913.

[18] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 575–589.

[19] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. 2010. Processor microarchitecture: An implementation perspective. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–116.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[21] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011).

[22] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[23] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. 2012. ILR: Where'd my gadgets go?. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 571–585.

[24] Tim Kornau et al. 2010. *Return oriented programming for the ARM architecture*. Ph.D. Dissertation. Master's thesis, Ruhr-Universität Bochum.

[25] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2018. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. 81–116.

[26] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. 2010. Defeating return-oriented rootkits with" return-less" kernels. In *Proceedings of the 5th European conference on Computer systems*. 195–208.

[27] J Ligatti, M Abadi, M Bidiu, and U Erlingsson. 2005. Control Flow integrity. In *Proceedings of the 12th ACM Conference on Computer and communications security*.

[28] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 941–951.

[29] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*. 49–58.

[30] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1406–1418.

[31] Hilmi Ozdoganoglu, TN Vijaykumar, Carla E Brodley, Benjamin A Kuperman, and Ankit Jalote. 2006. SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Trans. Comput.* 55, 10 (2006), 1271–1285.

[32] 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.

[33] Team PaX. 2003. PaX address space layout randomization (ASLR). *http://pax. grsecurity. net/docs/aslr. txt* (2003).

[34] Manish Prasad and Tzi-cker Chiueh. 2003. A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks.. In *USENIX Annual Technical Conference, General Track*. 211–224.

[35] Jonathan Salwan. 2011. ROPgadget–Gadgets finder and auto-roper. *Arzon, France, Tech. Rep., Mar* (2011).

[36] David Seal. 2001. *ARM architecture reference manual*. Pearson Education.

[37] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*. 552–561.

[38] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*. 298–307.

[39] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza 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.

[40] Eugene H Spafford. 1989. The Internet worm program: An analysis. *ACM SIGCOMM Computer Communication Review* 19, 1 (1989), 17–57.

[41] W Richard Stevens and Stephen A Rago. 2008. *Advanced programming in the UNIX environment*. Addison-Wesley.

[42] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2018. Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 624–637.

[43] 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)*. 941–955.

[44] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 41–50.

[45] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 719–732.

[46] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 559–573.

[47] Mingwei Zhang and R Sekar. 2013. Control flow integrity for {COTS} binaries. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 337–352.