Zipper Stack: Shadow Stacks Without Shadow

Jinfeng Li, Liwei Chen, Qizhen Xu, Linan Tian, Gang Shi, Kai Chen, and Dan Meng

Institute of Information Engineering, Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
{lijinfeng,chenliwei,xuqizhen,tianlinan,shigang,chenkai,mengdan}@iie.ac.cn

Abstract

Return-Oriented Programming (ROP) is a typical attack technique that can exploit return addresses to repeatedly abuse existing codes ending with return instructions. Most of the current return address protecting mechanisms (also known as the Backward-Edge Control-Flow Integrity) can work only in limited threat models. For example, the attacker cannot control the whole memory, or the attacker have no knowledge of a secret key or random values.

This paper presents a novel, lightweight mechanism protecting return addresses, Zipper Stack, which hashes all return addresses by a chain structure. This innovative design can defend against the most powerful attackers who have full control over the program’s memory and even know the secret key of the hash function. This threat model is stronger than them in the relevant works. At the same time, it produces very low performance overhead. We implemented Zipper Stack by extending the RISC-V instruction set architecture and the evaluation shows that the performance overhead of Zipper Stack is only 1.86% (vs a Shadow Stack on the same platform costs 2.36%). Additionally, the lightweight nature of Zipper Stack makes it practicable for deployment with minimal modifications on the system. We only need two registers and a hash module, no need to make any changes to the memory allocation and page attributes. Thus, we think Zipper Stack is suitable for actual deployment.

1 Introduction

In the exploitation of memory corruption bugs, return address is one of the most widely exploited vulnerable points. On the one hand, code-reuse attacks (CRAs), such as ROP [8] and ret2libc [33], are state-of-the-art attack techniques. They perform malicious behaviours by chaining short sequences of instructions which end with a Return via corrupted return addresses. These attacks require no code injection so they could bypass non-executable memory protection. On the other hand, the most widely exploited memory vulnerability, stack overflow, is also exploited by overwriting the return address. Both CRAs and stack smashing attacks rely on tampering with the return addresses.

In order to protect the return addresses, quite a few methods were presented, such as Stack Protector (also known as Stack Canary) [11, 16, 46], Address Space Layout Randomization (ASLR) [39], Shadow Stacks [9, 36, 12, 30], Control Flow Integrity (CFI) [2, 3], and Cryptography-based CFI [26, 31]. However, they have encountered various problems in the actual deployment.

Stack Protector and ASLR rely on random values (cookies or memory layout) that attackers cannot disclose. Both methods are used on deployment in practise. However if there is a memory leak, both methods will fail [37]: the attacker can overwrite with a same value to bypass the Stack Protector, and de-randomize the ASLR to bypass it. Some works have proved that they can be stably bypassed in some circumstances [34, 25, 20], such as BROP [6]. Even if there’s no information leaking, some approaches could still bypass ASLR and perform CRAs [32].

Shadow Stack is a direct mechanism which records all return addresses in a protected stack and checks them when returns occur. It has been developed via both compiler-based and instrumentation-based approaches [13, 12]. In recent years, commercial hardware support has also emerged [21]. But Shadow Stack relies heavily on the absolute security of the protected memory area, which is difficult to guarantee in actual deployment. Some designs of Shadow Stack utilize the ASLR to protect the Shadow Stack, however, they cannot thwart the attacks contains any information disclosure [12]. Since most methods that bypass ASLR [34, 25] are effective against this type of defense. Other designs use page attributes to protect the Shadow Stack, for instance, CET [21]. However, defenses that rely on page attributes, such as NX (no-execute bit), have been bypassed by various technologies in actual deployment [22]: a single corrupted code pointer to the function in the library (via a JOP/COOP attack) may change the page attribute and disable the protection. Some historical attacks have also proved the page
attributes are noneffective under certain circumstances. In light of these findings, we think that mechanisms which do not rely on memory security are more reliable and imperative.

Cryptography-based protection mechanisms have also been proposed, which are based on hash authentication. These methods hash the return addresses and authenticate them before returns. These mechanisms do not rely on memory security since the attacker cannot generate new correct hashes without the secret key. But they are facing other problems: replay attacks (which reuse the existing code pointers and the hash of them), high performance overhead and security of keys. A simple defence using hash cannot thwart the replay attacks, so some designs introduce extra complexity to avoid the problem (such as add a nonce or add stack pointer into the hash input), and the effect is not ideal. Besides, the performance overhead of these methods is also not negligible, since they use a cryptographic hash and then save the result into memory. Both hash calculations and memory accesses consume a lot of cycles in CPU. The security of the secret key is also a challenge, since once the secret key was leaked, the whole defend became noneffective. Theses works assume that there is no hardware attack or secret leak from kernel. In the real world, however, the secret key can be leaked by a side channel attack or an attack on context switching in kernel. Therefore, we should go further over these assumptions, and develop countermeasures resisting secret leak.

In this paper, we propose a concise and novel method to protect return addresses, Zipper Stack, which uses a lightweight hashing algorithm to protect the return addresses and the hashes of return addresses together. In the design of Zipper Stack, all addresses and hash values are in a chain. The newest hash value is generated by hashing the newest return address and the previous hash value. The previous hash value was generated by hashing the return address and hash value prior to the previous one, as the Figure 1 shows. So the newest hash value is calculated from all the former return addresses in the stack, although it is just generated by computing the newest address and the previous hash value. Zipper Stack minimizes the amount of state requiring direct protection: only the newest hash value needs to be protected from tampering. Without tampering the newest hash value, an attacker cannot tamper with any return address because he cannot tamper the whole chain and keep the relation.

Zipper Stack avoid the problems of Shadow Stack and Cryptography-based protection mechanisms. Compared with Shadow Stack, it does not rely on the security of any memory area. Consequently, the attacks that modify both the shadow stack and main stack cannot work in Zipper Stack: Zipper Stack uses the hash to verify the return address, not the copy. It also reduces the complexity of the design. Compared with other cryptography-based mechanisms, Zipper Stack can resist against replay attacks itself (because the same return address will not share the same hash value), and will not fail even if the secret key was leaked (because the newest hash cannot be tampered with, even if the attacker knows the secret key, he still needs to find hash collision for each corrupt return address, see Section V). In terms of efficiency, Zipper Stack performs even better. Our design is more suitable for parallel processing in the CPU pipeline, which avoids most performance overhead caused by the hash calculation and memory access. The performance overhead of Zipper Stack with hardware support based on Rocket Core (RISC-V CPU) is only 1.86% based on our experiments (versus a hardware based Shadow Stack costs 2.36%). Therefore, we think that Zipper Stack is a better alternative to Shadow Stack.

The neat design of Zipper Stack solved three challenges: First, it avoided the significant runtime overhead that most cryptography-based mechanisms suffer. Since the newest hash could be updated in parallel with normal execution. In our hardware implementation, most instructions that contain a hash take only one cycle (See Section 6). Second, it utilized the LIFO order of return addresses to minimize the amount of state requiring direct protection. In general, a trust root authenticating all the data could help us to defend against replay attacks or attacks that contain secret key leaks (which most current return address protection method cannot). While in Zipper Stack, the authentication uses the newest hash, at the same time, the hash is a dynamic trust root itself. So it gets better security without extra overhead. Third, previous methods protect each return address separately, so any one could be attacked. Zipper Stack, however, could connect all the return addresses together, leverage the prior information to increase the bar for attackers.

In order to demonstrate the design and evaluate the performance, we implement Zipper Stack in three deployments corresponding to three situations: a) Hardware approach, which is suitable when hardware support of Zipper Stack is available. b) Customized compiler approach, which is suitable when hardware support is not available but we can recompile the programs. c) Customized ISA approach, which is suitable when we cannot recompile the programs but we can alter the function of CALL/RET instructions. Ideally, hardware approach is best - it costs lowest runtime over-
head. So we mainly discuss this approach in this paper. The other two approaches, however, are suitable in some compromised situation. In hardware approach, we instantiated Zipper Stack with a customized Rocket Core (a RISC-V CPU) on the Xilinx Zynq VC707 evaluation board (and a hardware based Shadow Stack as a comparison). In customized compiler approach, we implemented Zipper Stack in LLVM. In customized ISA approach, we use Qemu to simulate the modified ISA.

Contributions. In summary, this paper makes the following contributions:

1. **Design:** We present a novel, concise, efficient return address protection mechanism, called Zipper Stack, which could protect return addresses against the attackers have full control of all the memory and know the secret keys, with no significant runtime overhead. Consequently, we analyze the security of our mechanism.

2. **Implementation:** To demonstrate the benefits of Zipper Stack, we implemented Zipper Stack on FPGA board, and a hardware-based Shadow Stack as a comparison. In order to demonstrate the potential of Zipper Stack to be further developed, we also implemented it in LLVM and Qemu.

3. **Evaluation:** We quantitatively evaluated the runtime performance overhead of Zipper Stack, which is better than existing mechanisms.

The remainder of this paper is organized as follows:

In Section 2, we introduce the background. In Section 3, we define our threat model. In Section 4, we introduce Zipper Stack in details. Then we discuss the security and performance overhead in Section 5. We introduce our implementations and evaluate them in Section 6 and Section 7.

## 2 Background and Related Work

### 2.1 ROP attacks

Return Oriented Programming (ROP) [33, 8] is the major form of code reuse attacks. ROP makes use of existing code snippets ending with return instructions called gadgets to perform malicious acts. In ROP attacks, the attackers link different gadgets by tampering with a series of return addresses. An attack is usually made up of multiple gadgets. At the end of each gadget, a return instruction links the next gadget via the next address in the stack. The defenses against ROP mainly prevent return instructions from using corrupted return addresses, or randomize the layout of the codes.

### 2.2 Shadow Stack and SafeStack

Shadow Stack is a typical technique to protect return addresses. Shadow Stack saves the return addresses in a separated memory area, and checks the return addresses in the main stack when returns. It has been developed in both compiler-based and instrumentation-based approaches [9, 35, 12, 30, 28, 27, 15, 29]. SafeStack [23] is a similar way, which moves all the return addresses into a separated stack instead of backs up the return addresses. SafeStack is now implemented in LLVM as a component of CPI [40]. Both methods are used as a part of the backward-edge Control-Flow Integrity and provides satisfactory security in certain threat model. These techniques need extra memory space to store copies of the return addresses, obviously, this area needs to be protected. In some implementations it depends on ASLR or modifications on the page attributes.

A separated stack mainly brings about two problems: One problem is that, the shadow stack is protected separately, which means that additional memory allocations are necessary and, in addition, special designs are required to protect it. Another problem is that, it requires the shadow stack to be perfectly protected, which is impractical. As the structure of shadow stack is simply the copies of return addresses, it is very fragile once the attacker is able to modify its memory area. For example, in Intel CET, the protection of shadow stack is provided by a new page attribute. But the similar approach in DEP is easily bypassed by a variety of methods modifying the page attribute in real-world attacks [22]. ASLR is also bypassed in real-world attacks which other implementations rely on. Some mechanisms use other data structure like hash tree (Merkle tree) to protect the shadow stack with a small trusted source (the root of the tree), e.g. [29]. However, the structure need several hash calculations to reach the trusted source, which costs more overhead.

In Zipper Stack, both the problems are solved: even if the attacker has full control over the program’s memory, he is not able to perform ROP attacks, since Zipper Stack do not rely on the security of any memory area; and Zipper Stack does not require a separated stack, and requires fewer modifications on compilers, OS, or hardware.

### 2.3 CFI

Control Flow Integrity (CFI), which first introduced by Abadi et. al. [2, 3], has been recognized as an important low-level security property. In CFI, runtime checks are added to enforce that jumps and calls/rets land only to valid locations that have been determined ahead of execution. CFI can protect from code reuse attacks (CRAs). Most of these CFI approaches rely on information analyzed from assembly codes or higher level, up to source code [7, 48, 42, 35].

The security and performance overhead of different implementations differ. Fine-grained CFI approaches will introduce significant overhead. However coarse-grained CFI has lower performance overhead but enforces weaker restrictions, which is not secure enough. In addition, Control-Flow Graphs (CFGs), which fine-grained CFI bases on, are constructed by analyzing either the disassembled binary or
source code. CFG cannot be both sound and complete, so even if efficiency losses are not mainly considered, CFI is not a panacea for code reuse attacks [14]. Due to the above reasons, CFI is not widely deployed on real systems now.

2.4 Crypto-based CFI

In order to optimize the CFI, some implementations also introduce cryptography methods to solve problems such as inaccurate static analysis: CCFI [26], RAguard [49]. Most of these methods are based on hash: the protected key pointers including return addresses are hashed and checked before use. But all of these methods rely heavily on secrets (hash challenges or keys). In order to ensure the confidentiality of the secrets, they must use a high-secure algorithm, which in turn, brings tremendous performance overhead. In Zipper Stack, the performance is much lower than previous methods. Another problem of these mechanisms is about replay attacks, the attackers can perform replay attacks by reusing the existing values in the memory. Some implementations add stack pointer such as RBP into the input of the hash function to avoid the problem, but it’s not ideal. Fortunately, Zipper Stack can naturally resist replay attacks. Besides, in these mechanisms, once the secret was figured out or violently cracked, the building of attacks becomes easy. However, Zipper Stack increases the difficulty of counterfeiting control flows: even if the attacker gets the secret, he still needs to find enough hash collisions to bypass the verification.

2.5 Randomization

Randomization-based approaches have also been proposed to mitigate code reuse attacks. Address Space Layout Randomization (ASLR) [39] and Stack Protector [11, 10] (also known as Stack Canary) are widely used among them. Address Space Layout Randomization (ASLR) performs by randomizing base addresses of code segments. In ROP attacks, attackers have to know the address of each gadget to launch the attack. Thus, ASLR seems effective in preventing ROP attacks. However, like most approaches that rely on randomization, ASLR can be bypassed by leveraging brute-force attacks or information leakage attacks.

Stack Protector inserts a random value above the return address. When stack overflows, this random value is changed, thus a potential attack has been detected. But Stack Protector can only work towards stack smashing attacks. In addition, when an attacker can read the stack, it can be bypassed by overwriting the random value with the same value.

3 Threat Model

In this paper we assume that a powerful attacker has the ability to read and overwrite arbitrary areas of memory. He tries to perform ROP (or ret2lib) attacks. This situation is very common - for example, a controllable pointer out of bounds could help the attacker to acquire the capability. Reasonably, the attacker cannot alter the value in the dedicated registers (called Top and Key registers in our design), since these registers cannot be accessed by general instructions. The attacker in our assumption is more powerful than all previous works. The Shadow Stacks assume that the attackers cannot locate or overwrite the shadow stack, which is part of the memory. In our work, we do not need that assumption, which means we could defend against more powerful attacks.

4 Design

In this section, we elaborate on the design of Zipper Stack in detail. Here, we take the hardware approach as an example. The design of other approaches is similar.

4.1 Overview

Figure 2. Overview of Zipper Stack

In hardware approach, we need hardware support and the modification of memory layout.

Figure 2 shows the overview of the hardware in Zipper Stack: Zipper Stack needs two dedicated registers and a hash module in the CPU, and it requires no hardware modification of the memory. The registers include the Top register holding a hash value (Nh bits) and the Key register holding a secret key (Ns bits). Both the registers should be initialized to random numbers at the beginning of a process and they cannot be read nor rewritten by attackers. The secret key will not altered in the same process. Therefore, we temporarily ignored this register for the sake of simplicity in the following. Assuming that the width of return addresses is Na, the hash module should perform a cryptographic hash function with an input bit width of Na + Nh and an output bit width of Nh. In addition, this hash function should include a random challenge (Key register) to avoid becoming a fixed function.

We now turn to the memory layout of Zipper Stack. In Zipper Stack, all return addresses are bound to a hash value, as shown in Figure 3. The novelty is, the hash value is not generated from the address bound with itself, but from the previous return address with the hash value bound with that address. This connection keeps all return addresses and hash values in a chain. In order to maintain the structure, the top one, namely the last return address pushed into the stack, is
hashed together with the previous hash and the output hash value is saved into the Top register; while the bottom one, i.e. the first return address pushed into the stack, is bound to a random number (exactly the initial value of Top register when the program begins).

4.2 Operations

Next, we describe how Zipper Stack works with return addresses in the runtime, i.e., how to handle the Call instructions and the Return instructions. As Figure 4 shows.

Call: In general, the Call instructions perform two operations. First, push the address of the next instruction into stack. Then, set the PC to the call destination. While in Zipper Stack, the Call instructions become slightly more complicated and need three steps:

1. Push the Top register along with the return address into main stack;
2. Hash the Top register along with the return address into a new hash value and save the new value into the Top register;
3. Set the PC to the call destination.

Return: In general, the Return instructions also perform two operations. First, pop the return address from the stack; second, set the PC to this address. Correspondingly, in Zipper Stack, Returns also become a little more complicated, including four steps.

1. Pop the return address and the previous hash value from the main stack, and use the same hash function before to hash them into a new hash value for check;
2. Check whether this new hash value is equal to the current hash value in the Top register. If not, raise an exception (which means an attack).
3. Save the hash value poped from the stack into the Top register.
4. Set the PC to the return address poped from the stack.

Figure 4 shows the process of CALL and RET in Zipper Stack. We omit the normal operations about the PC and return addresses.

The core idea of Zipper Stack is to use a chain structure to link all return addresses together. Based on this structure, we only need to focus on the protection and verification of the top of the chain instead of protecting the entire structure. Just like a zipper, only the slider is active, and when the zipper is pulled up, the following structure automatically bites up. Obviously, protecting a hash value from tampering is much easier than protecting a series of hash values from tampering: Adding a special register in the CPU is enough, and there is no need to protect a special memory area.
### 4.3 Setjump/Longjump and C++ Exceptions

In most cases, the return addresses are used in a LIFO order (last in, first out). But there are exceptions, such as setjump-/longjump and C++ exceptions. Consequently, most mechanisms protecting return addresses suffer from the Setjump-/Longjump and C++ Exceptions. Both Setjump/Longjump and C++ Exceptions save and restore the context between different functions. The main task of them is stack unwind. So we are only concerned with the Top register and the return addresses, since other operations in the stack unwinding have nothing to do with us.

The return addresses in the stack will not encounter any problem, since Zipper Stack does not alter either the value nor the position of return addresses. The only problem is how to restore the Top register. The solution is quite simple: backup the Top register just like backup the stack pointer or other registers (in Setjump/Longjump save them in the jump buffer, similar in C++ Exceptions). When we need to Longjump or handle an exception, restore the Top register just as restoring other registers. The chain structure will remain tight. However, the jump buffer is in the memory, so this solution exposes the Top register (since the attacker is able to write on arbitrary areas of memory) and leaves an opportunity to overwrite the Top register. So additional protection is a must. There are several ways to fix it:

1. Hash the Top register and other values in the jump buffer into a checksum and add it into the jump buffer. Check it when long jump.
2. Similar to the first one, encrypt the Top register and other values in the jump buffer and decrypt them when long jump.
3. In C++ exceptions, the exception structures are in a linked list, so we can use another Zipper-Stack-like structure to protect all the exception structures. This requires extra Top and Key registers.
4. Design a new system call to save the values in somewhere out of the process memory space (in kernel). In the first and second way, we need to protect other values along with Top register, otherwise the attacker may perform replay attacks. Besides, the load/store instructions for Top register are special here, and only used in Setjump/Longjump.

These instructions are always followed by the checking, so the attacker is not able to reuse these code to modify the Top register without check. We think the most secure and ideal solution is the last one, but it may lead to extra runtime overhead since it use system calls. In our implementation, we use the first way to protect the jump buffer in our prototype, in this way, we could reuse the hash module. How to protect the jump buffer is not the focus of this paper, the jump buffer is a natural vulnerability since it can overwrite the registers via memory. The above solutions provides satisfactory protection on Top register, however, our design will benefit from any design that protect the context switching.

### 5 Analysis and Comparison on Designs

In this section, we will analysis Zipper Stack in terms of security, performance and system complexity, and compare them with other mechanisms.

#### 5.1 Security Analysis

Now, we discuss the security of Zipper Stack. The challenge for the attacker is clear: how to tamper with the memory to make the fake return address be used and bypass our check? We list the defence effect of different methods of protecting return addresses in the face of different attackers in Table 1. The table shows that Zipper Stack has higher security than the Shadow Stack and other cryptography-based protection mechanisms.

#### 5.1.1 Direct Overwrite

First, we consider direct overwrite attacks. Here, the overwrite includes the stack overflow and arbitrary overwrite. In the previous cryptography-based methods, the adversary cannot know the key and calculate the correct hash value, so it is secure. But we go further that the adversary may...
steal the key and can calculate the correct hash. As Figure 5 shows, in order to tamper with any return address structure and bypass the check (let’s say, Return Address N), the attacker must bypass the pre-use check. Even if the attacker has stolen the key, the attacker needs to tamper with the hash value which is used to check the return address, i.e. the hash value stored beside Return Address N+1. Since the hash value and the return address are protected by the hash algorithm together, the attacker has to modify the hash value stored beside Return Address N+2 in order to tamper with the hash value bound to Return Address N+1. And so on, the attacker has to modify the hash value at the top, which is in the Top register. As we have assumed, the register is secure against tampering. As a result, an overwrite attack won’t work.

![Figure 5. Direct Overwrite Attack: The solid lines indicate the protect relation of the hash values, and the dotted lines show the order that the attacker should overwrite in.](image)

5.1.3 Brute-force Attack

Next, we discuss the security of Zipper Stack in the face of brute force. Here we consider the attacks which read all related data in the memory, guess the secret constantly, and finally construct the attack. In the cryptography-based approaches, security is closely related to the entropy of the secret. Here in Zipper Stack, the entropy of the secret is the bit width of Key register (Ns), which means the attacker needs to guess the correct Key register in a space of $2^{Ns}$.

The difference between Zipper Stack and other cryptography-based approaches is, since the ciphertext-hashes can be tampered with in the memory, other approaches will fail to protect control flow once the attacker knows the secret, but Zipper Stack will not. Because even if the attacker knows the value in Key register (secret key), the Top register cannot be tampered with. If an attack contains N gadgets, the attacker needs to find N hash collisions whose input contains the ROP (or ret2lib) gadget addresses in order to use the gadgets and bypass the check. Considering an ideal hash function, one collision will take about $2^{Nh-1}$ times of guesses on average. So an attack with N gadgets will take $N \cdot 2^{Nh-1}$ times of guesses on average even if the Key register is leaked. The total number of guesses is (guessing Key value and the collisions) $2^{Nh-1} + N \cdot 2^{Nh-1}$. And more unfortunate for the attackers: under a certain probability $(1 - (1 - 1/e)^N)$ for an attack contains N gadgets), the valid hash collision does not even exist. Take an attack contains 5 gadgets as example, the possibility that the hash collision does not exist is around 90%, and the possibility grows as the N grows.

5.2 Performance Analysis

In Zipper Stack, the performance overhead comes mainly from two aspects: memory access and cryptographic calculations. Here we enumerate the comparison with Shadow Stack and other Cryptographic-based mechanisms, as shown in Table 2.

| Mechanism                  | Memory R/W | Calculation | Overhead |
|----------------------------|------------|-------------|----------|
| Zipper Stack non-compressed| Hash Value | Hash (Parallel) | Medium   |
| Zipper Stack compressed    | None       | Hash (Parallel) | Low      |
| Shadow Stack               | Backup Address | None | Low      |
| Hash & Check               | Hash Value | Hash | High     |

Memory Access. Zipper Stack requires no more memory access than other algorithms and has the best locality, which is important when cache is limited. In addition, in most circumstances, not all the bits of return addresses are used to address the code. Consequently we can replace the leading zeros with the hash value. In the remainder of this paper, we

---

1. The same gadget addresses won’t share the same collision, because the hash values bound to them differ.

2. The hash function is a $(Nh+Na)$-bit-to-$Nh$-bit function, so choosing a gadget address will determine $Na$ bits of input, which means there are $2^{Nh}$ optional values. On average, there is one hash collision in the values, since every $2^{Nh-Na}/2^{Nh} = 2^{Na}$ inputs share the same hash value. Because of our special algorithm, this is not a birthday attack nor an ordinary second preimage attack, but a limited second preimage attack.

3. Although Stack Guard seems require less memory access, but it protect less return addresses. Considering “memory access per protected address” both methods equal.

---
call this *Compressed Structure*. This design is used in previous methods, such as [31]. With the Compressed Structure, there is no extra memory access.

**Calculation.** With hardware support, Zipper Stack also has an advantage on the overhead of hash calculation compared with other cryptography-based mechanisms. When a Call instruction occurs, a hash value and return address (or an encrypted return address) need to be written to memory. Other mechanisms have to wait for the calculation to be completed before storing the result to memory. While in Zipper Stack, the value storing to memory is already in the Top register, and the new hash value is calculated and stored into Top register. So the calculation and the memory access can be handled in parallel. Figure 6 is a schematic of how Zipper Stack differs from other methods in processing hash values.

In our hardware implementation, we use compressed structure. The hash is stored together with the return address. So in most scenario, our new instruction consumes only one cycle, like a nop. The evaluation also shows that the hash calculation hardly hinders other operations and has only a slight impact on runtime overhead. See Section 6.3.

In our hardware implementation, we use compressed structure. The hash is stored together with the return address. So in most scenario, our new instruction consumes only one cycle, like a nop. The evaluation also shows that the hash calculation hardly hinders other operations and has only a slight impact on runtime overhead. See Section 6.3.

![Figure 6: Difference in Pipeline](image)

### 5.3 System Complexity

In this section, we discuss how to use Zipper Stack in real deployments, i.e., what changes need to be made to the compiler, the CPU, and the operating system. Additionally, we compare the changes with them in Shadow Stack. We take Control-flow Enforcement Technology (CET) [21] for example, which includes a formal shadow stack implementation. In CET, the shadow stack is protected from tampering through the page table protections such that regular store instructions cannot modify the contents of the shadow stack. To provide this protection the page table protections are extended to support an additional attribute for pages to mark them as "Shadow Stack" pages.

**Page Attribute** In CET, a new page attribute leads to adaptation problems at all levels. Except for the hardware support in CPU, kernel support is also needed to make use of the attribute, and to insure the security of the protected pages. In addition, the compiler also needs modifications, and related libraries should be re-compiled. Zipper Stack will not encounter these problems since it does not need a new page attribute.

**CPU** Both Zipper Stack and Shadow Stack need special registers (Top and Key in Zipper Stack vs Shadow Stack Pointers in CET), which require hardware modification. In Zipper Stack, we need an arbitrary hash module (The good news is, similar modules are generally present in modern CPUs, and we can reuse them directly); in Shadow Stacks, we need a new page attribute support.

**Compiler and OS** In operating system, both methods need an initialization before the beginning of each program. In addition, these registers also need to be saved and restored when the processes are switched. After initialization, the two are similar, but Zipper Stack has a simpler initialization: In Zipper Stack, the special registers should be initialized as random numbers; while in Shadow Stacks, we have to allocate a special memory space and initialized the Shadow Stack Pointer.

To sum up, we think Zipper Stack requires less modifications on the whole system than Shadow Stack.

### 6 Implementation

In this section, we introduce our implementation. We mainly talk about the hardware approach first. In hardware approach, we implemented a prototype of Zipper Stack by modifying the Rocket Chip Generator [43] and customized the RISC-V instruction set accordingly. We also added a hash module, several registers and several instructions into the core. Whereafter, we modified the tool chain including the compiler and the library glibc. Besides, we implemented a similar Shadow Stack for compare. At last, we introduce the customized compiler approach (LLVM implementation) and customized ISA approach (Qemu implementation).

#### 6.1 Overview

In Rocket core, we added a Top register and a Key register, which correspond to those designed in the algorithm. These two registers cannot be loaded/stored via normal load/store instructions. At the beginning of a program, the Key and Top register are initialized by random values.

In RISC-V architecture, a CALL instruction will store the next PC, i.e. return address, to the *ra* register, and a RET instruction will read the address in the ra register and jump to the address. Consequently two instructions were added in our prototype: PAC (process after call), PBR (process before return). As their name implies, they will perform as a Zipper Stack’s CALL/RET together with a normal CALL/RET.

For the sake of simplicity, we use a compressed structure. In RISC-V architecture, the return address in register *ra*, so we put the return address and the hash value together into *ra*. In the current Rocket core, only lower 40 bits are used
to store the address. Therefore, we use the upper 24 bits to hold the hash value. Correspondingly, our Top register is 24 bits. And our Key register is 64 bits.

Our new instructions will update a hash (PAC after a CALL) or check and restore a hash (PBR before a RET). When a PAC instruction is executed, the address in ra (only lower 40 bits) along with the old hash in the Top register will be hashed, the result (new hash) will be stored in Top register. The old hash is stored to the higher 24 bits in the ra register (the lower 40 bits remain unchanged). Correspondingly, when a PBR instruction is executed, the ra register (including the hashes and address) is hashed and compared with the Top register. If the values match, the hash value in ra (higher 24 bits) is restored into the Top register, and the higher 24 bits in ra is restored to zero. If the values do not match, an exception will be raised (which means an attack). The following piece of assembly code shows how a function is protected in Zipper Stack.

```assembly
  call fun
  ...          
  fun:
    + pac
    ... # function body
    ...  
    + pbr
    ret
```

### 6.2 Hash Module

Next, we added a hash module in the Rocket Core. Here, we use Keccak [5] (the algorithm used by Secure Hash Algorithm 3 (SHA-3)) as the hash function. In our hardware implementation, the arguments of Keccak are as follows:

\[
\begin{align*}
  l &= 4, r = 256, c = 144
\end{align*}
\]

The main difference between our implementation and SHA-3 is that we use a smaller \( l \): in SHA-3, \( l = 6 \). We made this change because we think the capacity of Keccak state in SHA-3 is a little wasteful here. And this change will also slightly reduces the latency of each operation (the number of absorb rounds is reduced from 24 to 20). This hash module will take 20 cycles for one hash calculation normally and it uses 793 LUTs and 432 flip flops. As a comparison, the multiplier costs 1686 LUTs and 214 flip flops. So the hash module incurs less area overhead than a multiplier. In fact, for a CPU that already contains a hash module, we don’t need extra area overhead.

### 6.3 Pipeline

The pipeline in Rocket Core is a 5-stage single-issue in-order pipeline. In order to reduce possible performance losses, the hash calculations are processed in parallel. If the next instruction that uses a hash arrives after the previous hash finished, the pipeline will not stall. Figure 7 is a pipeline diagram of two instructions. In Figure 7, IF, ID, EX, MEM, WB stand for fetch, decode, execute, memory and write back stages. UD/CK stands for updating Top register/checking hash values. As the figure shows, if the next PAC/PBR instruction arrives after the Hash calculation, only one extra cycle is added to the pipeline, which is equivalent to inserting a nop. It is worth noting that the WB stage in PAC/PBR do not rely on the finish of UD/CK stage: in write back stage it only write the ra register, and the value do not rely on the current hash calculation. Fortunately, most functions require more cycles than 20. So in most cases, the PAC/PBR instruction takes only one cycle. Only frequent calls and returns, such as a small function which recurs itself, will significantly increase the performance loss. In the next section, we tested these quantitatively.

### 6.4 Customized Compiler

To make use of the new instructions, we also customized the riscv-gcc. It is noteworthy that, if a function will not call any function, ra register will not be spilled to the stack. So we only add the new instructions when the ra register is saved into/restored from the stack (rather than all calls and returns). The modification on riscv-gcc is quite simple: Whenever we store a return address onto the stack, we add a PAC instruction; whenever we pop a return address from the stack, we add a PBR instruction. The following two pieces of assembly code show how our compiler use the new instructions.

```assembly
  # Modified if with Spilling ra:
  fun:
    + pac
        sd ra, 8(sp) # save ra
        ... # function body
        ld ra, 8(sp) # restore ra
    + pbr
    ret
```

---

4Term of Keccak algorithm.
6.5 Setjmp/Longjmp Support

To support Setjmp/Longjmp, we also modified the glibc in the RISC-V tool chain. We have only modified two points:

1. Declaration of the Jump Buffer: Add additional space for the Top register and Hash value.
2. Setjmp/Longjmp: Store/restore the Top register; Calculate/check the Hash value.

Our changes perfectly support Setjmp/Longjmp, which is verified in some benchmarks in SPEC2000, such as perlbench. These benchmarks will not pass without Setjmp/Longjmp support.

6.6 Optimization

In order to further reduce the runtime overhead, we also optimized the hash module. In our scenario, there are lots of repeated calculations. So we added a small cache (with a size of 4) to cache the recent hash results. If a new hash request can be found in the cache, the hash calculation will take only one cycle. This optimization slightly increased the complexity of the hardware, but significantly reduced runtime overhead. The cache incurs an area overhead of just 183 LUTs and 380 flip flops, while the runtime overhead is reduced by around 30% (see Section 7).

Obviously, we made very few changes in the original hardware, only a few registers and some logic modification for new instructions are added. Furthermore, there is no increase of the memory access.

6.7 A Comparable Hardware Based Shadow Stack

In order to compare with Shadow Stacks, we also implemented a hardware supported Shadow Stack on Rocket Core. We tried to be consistent as much as possible: We added two instructions that can back up or check the return address, and a pointer pointing the shadow stack. The compiler with Shadow Stacks inserts the instructions just like the way in Zipper Stack. The exception support is also similar. At the beginning of each program, we allocate a memory area to place the shadow stack.

6.8 Other Implementations

To facilitate the evaluation of security and compatibility, we also implemented Zipper Stack on Qemu. To demonstrate the potential of Zipper Stack in the compiler implementation and with different bit width, we also implemented Zipper Stack based on LLVM.

6.8.1 Qemu

We want to figure out if we change the logic of calls and returns (into them in Zipper Stack), is it possible to use the existing binaries without modification. In addition, in order to demonstrate Zipper Stack’s security, we customized the x86-64 instruction set, and used Qemu to simulate it. All the simulation is in the User Mode of Qemu 2.7.0.

The modification is quite concise, we add two registers and change the logic of Call and Ret instructions. As the algorithm designed, the Call instruction will push the address and the Top register, update the Top register with a new hash value, while Ret instruction will pop the return address and check the hash value. These can be achieved by modifying the intermediate representation (IR) in Qemu.

Here, we use a compressed structure. Since Qemu uses lower 39 bits to address the memory, we use the upper 25 bits to store the hash value. Correspondingly, the width of the Top register is also 25 bits. The Key register here is 64 bits. Both of them should be initialized by random numbers. We used SHA-3 as the hash function in this implementation. Since we did not change the stack structure, this implementation has good binary compatibility. Therefore it could help us to evaluate the security with real x86-64 attacks.

6.8.2 LLVM

Can we use the current x86-64 CPUs to perform Zipper Stack with a modified compiler? The answer is Yes. We implemented Zipper Stack algorithm based on LLVM 4.0 Function Pass. First we allocate two registers: we set the lower 64 bits of XMM15 as the Top register, and the XMM14 as the Key register. We modified the backend of the LLVM so as to forbid both registers to be used by anything else. At the beginning of a program, the Key and Top register are initialized by random numbers.

Next, our implementation leverages the AES-NI instructions on the Inter x86-64 architecture to minimize the performance impact of hash calculation. (In our experimental environment (Intel Core i7), accelerate instructions for SHA are not supported, so we use AES-NI instead.) We use the Key register as the round key of AES-NI, and use one 128-bit AES block as the hash input (64-bit address and 64-bit hash). The 128-bit result is truncated into 64-bit in order to fit our design.

In each function, we insert a prologue at the entry, and an epilogue before the return. In the prologue, the old Top register is saved onto stack, and updated to the new hash value of the current return address and the old Top register value. In the epilogue, the hash value in the stack and return
address are hashed again and compared with the hash value in the Top register. If it doesn’t match, an exception will be raised. Just as we introduced before.

In Linux ABI, unfortunately, the XMM registers are scratch registers, which means the shared libraries may alter the value in the Top register and Key register. So we also recompiled the shared libraries so as to keep them from using XMM14 and XMM15. However, this problem can be circumvented by using r15/r14 instead, since these registers are callee-save registers. The price is that performance loss increases (because r15/r14 are more often used in most programs).

In this implementation, the memory access is equivalent to that in the Shadow Stacks, but with better locality. Furthermore, we do not need to allocate new pages for a separate stack, which means in the real world, Zipper Stack consumes lower memory.

7 Evaluation

We evaluated Zipper Stack in two aspects: Runtime Performance and Security. We evaluated the performance overhead with the SPEC CPU 2000 on the FPGA board and the Dhrystone using RTL simulation. We also evaluated the performance overhead with different hash latency, to explore the impact of the hash latency on performance.

Besides, we also evaluate Zipper Stack in other aspects: a) Does our solution to the Setjmp/Longjmp work? b) If we only modify the Call and Ret instructions in the x86-64 ISA, and use the compression structure to maintain the stack layout, is it possible to maintain binary compatibility directly? c) The estimated performance overhead of Zipper Stack based on customized compiler (LLVM implementation).

7.1 Performance Overhead

7.1.1 SPEC CINT 2000

To evaluate the performance of Zipper Stack on RISC-V, we instantiated it on the Xilinx Zynq VC707 evaluation board and ran the SPEC CINT 2000 benchmark suite (due to the limited computing power of the Rocket Chip on FPGA, we chose the SPEC 2000 instead of SPEC 2006). The OS kernel is Linux 4.15.0 with support for the RISC-V architecture. The hardware and GNU tool-chain are based on freedom (commit cd9a525). All the benchmarks are compiled with GCC version 7.2.0 and -O2 optimization level. We ran each benchmark for 3 times.

Table 3 shows the results of Zipper Stack and Shadow Stack. The result shows that without optimization, Zipper Stack is slightly slower than Shadow Stacks (2.69% vs 2.36%); while with optimization (the cache), Zipper Stack is much faster than Shadow Stacks (1.86% vs 2.36%). To sum up, the runtime overhead of Zipper Stack is satisfactory (1.86%).

7.1.2 Performance Overhead with Different Hash Latency

We also tested the runtime overhead of different hash modules with different cycles, in order to clarify the relationship between the latency of hash operation and runtime overhead. Since we use Keccak algorithm, so different hash modules are 10/20/40/80 cycles, corresponding to 10-20 rounds of absorbing, and each round taking 1/2/4 cycles. A SHA-3 calculation will take 24 cycles, so we also tested the overhead of 24 cycles. The Figure 8 shows the results.

The results show that when the hash calculation is faster than 30 cycles, the runtime overhead is quite low; but when it is slower than 30 cycles, the overhead increases sharply with the increase of the cycles of the hash module. We speculate that this rule is due to the average number of cycles between adjacent two call/ret instructions. After one call/ret (pac/pbr), the hash module is busy. If the next call/ret instruction comes before the hash module finishing the calculation, the pipeline will stall to wait for the calculation. So the hash calculation cannot be handled in parallel if the hash cannot finish before next call/ret comes, as a result, the overhead is high. Otherwise, the runtime overhead is fairly low.

Besides, the results also show that our optimization effect is remarkable. Moreover, the higher the latency of the hash module, the more significant the optimization effect. A noteworthy piece of data is 24, 2.30%(Optimized). Which means, if we use SHA-3 as the hash function, the performance overhead is exactly 2.30%, since the SHA-3 will take exactly 24 cycles in our implementation.
Table 3: Result of SPEC 2000 on FPGA

| Benchmark | Baseline | Shadow Stack | Zipper Stack | Zipper Stack (optimized) |
|-----------|----------|--------------|--------------|-------------------------|
| 164.gzip  | 10923.10 | 10961.65 (0.35%) | 10960.60 (0.34%) | 10948.88 (0.24%) |
| 175.vpr   | 7442.48  | 7528.06 (1.15%) | 7490.49 (0.65%) | 7485.40 (0.58%) |
| 176.gcc   | 8227.93  | 8318.83 (1.10%) | 8348.99 (1.47%) | 8317.34 (1.09%) |
| 181.mcf   | 11128.67 | 11533.01 (0.22%) | 11183.31 (0.49%) | 11168.93 (0.36%) |
| 186.crafty| 10574.27 | 10942.89 (3.49%) | 10689.74 (1.09%) | 10692.53 (1.12%) |
| 197.parser| 8318.16  | 8577.67 (3.12%) | 8658.89 (4.10%) | 8544.72 (2.72%) |
| 252.eon   | 14467.81 | 15111.99 (4.45%) | 15519.98 (7.27%) | 15040.26 (3.96%) |
| 253.perlbmk| 7058.96 | 7310.78 (3.57%) | 7388.20 (4.66%) | 7342.20 (4.01%) |
| 254.gap   | 7728.56  | 7850.32 (1.58%) | 7926.10 (2.56%) | 7817.52 (1.15%) |
| 255.vortex| 13753.47 | 14738.06 (7.16%) | 14748.70 (7.24%) | 14644.70 (6.48%) |
| 256.bzip2 | 6829.01  | 6893.50 (0.94%) | 6954.81 (1.84%) | 6865.27 (0.53%) |
| 300.twolf | 11904.25 | 12044.16 (1.18%) | 11974.22 (0.59%) | 11917.37 (0.11%) |
| **Average** | 2.36% | 2.69% | 1.86% |

7.1.3 Cycles per New Instruction

To evaluate the cycles per new instruction (i.e., how many cycles does a pac/pbr instruction consume on average), we also used an RTL simulation provided by Rocket Chip Generator. We chose the high-performance, cycle-accurate C++ simulator to run the benchmark Dhrystone [47].

Table 4: Dhrystone on RTL Simulation

| Method       | Shadow Stack | Zipper Stack |
|--------------|--------------|--------------|
| Performance Overhead | 0.96% | 0.60% |
| Retired INS Increment | 4004 | 4004 |
| Total Cycles Increment | 8158 | 4592 |
| Cycles per INS | 2.04 | 1.15 |

After 10 runs, the Shadow Stack’s average runtime overhead on Dhrystone is 0.96% and Zipper Stack’s average runtime overhead is only 0.60%. In both methods, retired instructions increased by 4004, and the number of total cycles increased by 8158 in Shadow Stack and by 4592 in Zipper Stack. Which means, one instruction for Shadow Stack takes 2.04 cycles and one instruction for Zipper Stack takes only 1.15 cycles on average. Table 4 shows the comparisons.

This result shows that the performance overhead of Zipper Stack with hardware support is fairly low. And it proves that our inference in Section 6.3 is consistent with the actual situation: in most cases, a new instruction we added will take only one cycle.

7.2 Security Evaluation

7.2.1 Attack Tests

We tested some vulnerabilities and the corresponding attacks to evaluate the security of Zipper Stack. In these tests, we use Qemu implementation, because most attacks are very sensitive to the stack layout, a customized compiler (or just a compiler in different version) may lead to failures. Using Qemu simulation can keep the stack layout unchanged, avoid the illusion that the defense works which is actually because of the stack layout changes. In other words, because of our good binary compatibility on Qemu implementations, we can use some attacks without modification to evaluate the security of Zipper Stack.

We wrote a test suite contains 18 attacks and the corresponding vulnerable programs. Since Zipper Stack protects return addresses, each attack contains at least one exploit on return addresses, including stack overflow, ROP gadget or ret2lib gadget. The vulnerabilities include stack overflow and heap overflow. We listed as many combinations as possible. All attacks are detected and stopped (all of them will alter the hash value and cannot pass the check in the Return). These tests show that Zipper Stack is reliable.

7.2.2 Entropy Analysis

In this subsection, we summarize and analyze the entropy in various implementations and calculate their defensiveness in the face of brute force. The calculation method has already been deduced before (see Table 1). In Table 5, we list the Nh, Ns and the average number of times that a brute-force attack needs to try in each implementation. N represents the number of gadgets in an attack.

Based on the calculation, the security of Zipper Stack is satisfactory: the attacker need much more than $2^{53}$ attempts,
and the difficulty of the attacks increases sharply as the number of gadgets increases.

7.3 Other Results

7.3.1 Setjmp/Longjmp Support

In the SPEC CPU 2000, several benchmarks require setjmp/longjmp support, such as perlbmk. Based on our experiments, without our support of setjmp/longjmp, the benchmarks will fail. As long as the setjmp/longjmp is supported, the benchmarks passed. This proves that the method we mentioned in Section 4.3 perfectly supports setjmp/longjmp.

7.3.2 Compatibility Test

Here, we test the binary compatibility of Zipper Stack. It should be noted that this test is only valid for Qemu implementation. In the other two implementations, due to we have modified the compiler, we could use Zipper Stack as long as we recompile the source code, so there is no compatibility issue. The purpose of this test is: If we only modify the Call and Ret instructions in the x86-64 ISA, and use the compression structure to maintain the stack layout, is it possible to maintain binary compatibility directly?

We chose randomly 50 programs in Ubuntu (under the path /usr/bin) to test the compatibility in Qemu. 42 out of 50 programs are compatible to our mechanism. Most failures are due to the Setjmp/Longjmp, which we have not supported yet. So we think although there are some issues that need to be solved (such as the setjmp/longjump), Zipper Stack can be used directly on most existing x86-64 binaries.

7.3.3 Performance Evaluation in LLVM

To evaluate the performance of Zipper Stack on customized compiler, we run the SPEC CPU 2006 [19] compiled by our customized LLVM. It is worth noting that, the LLVM implementation use AES-NI instructions, not a standard hash algorithm, so performance results are estimated.

Table 6 shows the performance overhead of Zipper Stack on LLVM. Shadow Stack is reported to cost about 2.5-5% [38 [12], and we also implemented a Shadow Stack in LLVM 4.0 and run the SPEC CPU 2006 in the same way as Zipper Stack, which costs 3.09%. The performance overhead of Zipper Stack is 2.48% on average, which is slightly faster than Shadow Stack.

8 Future Work and Conclusion

In this paper, we proposed Zipper Stack, a novel algorithm of return address protection, which hashes all return addresses by a chain structure. It minimizes the amount of state requiring direct protection and costs very low performance overhead.

Through our analysis, Zipper Stack is an ideal way to protect return addresses, and we think it is a better alternative to Shadow Stack. We discussed various possible attackers and attacks in detail, concluding that an attacker cannot bypass Zipper Stack and then alter the return addresses. In most cases, Zipper Stack is more secure than existing methods. The simulation of attacks on Qemu also corroborates the security of Zipper Stack. Our experiment also evaluated the runtime performance of Zipper Stack, and the results have shown that the performance loss of Zipper Stack is very low. The performance overhead with hardware support based on Rocket Core is only 1.86% on average (versus a hardware based Shadow Stack costs 2.36%). We also discussed the changes that need to be made in the actual deployment, concluding that the changes are simple enough. Thus, the proposed design is suitable for actual deployment.

Zipper Stack also has the potential to be implemented with different parameters on various platforms, and to keep binary compatibility with the existing programs using x86-64 instruction set. We will go further on it, apply Zipper Stack more widely and make further optimizations.

---

Table 5: Entropy Calculation

| Platform | Nh | Ns | Offline Tries |
|----------|----|----|---------------|
| LLVM     | 64 | 128| $2^{127} + N \times 2^{63}$ |
| Qemu     | 25 | 64 | $2^{63} + N \times 2^{24}$ |
| RISC-V   | 24 | 64 | $2^{63} + N \times 2^{23}$ |

---

Table 6: Performance Overhead on LLVM

| Benchmark | Origin   | Shadow Stack | Zipper Stack |
|-----------|----------|--------------|--------------|
| 401.bzip2 | 778.04s  | 789.13s (1.43%) | 792.95s (1.92%) |
| 429.mcf   | 372.85s  | 403.95s (8.34%) | 393.71s (5.59%) |
| 445.gobmk | 562.39s  | 582.40s (3.56%) | 583.15s (3.69%) |
| 456.hmmer | 1004.49s | 1007.82s (0.33%) | 988.10s (-1.63%) |
| 458.sjeng | 680.83s  | 700.11s (2.83%) | 708.84s (4.11%) |
| 462.libquantum | 509.09s | 511.58s (0.49%) | 508.27s (-0.16%) |
| 464.h264ref | 932.75s | 976.22s (4.66%) | 968.38s (3.82%) |
| **Average** | **3.09%** | **2.48%** | |
References

[1] A. OU, A. WATERMAN, Q. N. D. B., AND DABBELT, P. Riscv linux port, 2015.

[2] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity. In Proceedings of the 12th ACM Conference on Computer and Communications Security (New York, NY, USA, 2005), CCS ’05, ACM, pp. 340–353.

[3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity Principles, Implementations, and Applications. ACM Trans. Inf. Syst. Secur. 13, 1 (Nov. 2009), 4:1–4:40.

[4] BELLARD, F. QEMU, a fast and portable dynamic translator. In USENIX Annual Technical Conference, FREENIX Track (2005), pp. 41–46.

[5] BERTONI, G., DAEMEN, J., PEETERS, M., AND VAN ASSCHE, G. Keccak sponge function family main document. Submission to NIST (Round 2) 3, 30 (2009).

[6] BITTAU, A., BELAY, A., MASHITZADEH, A., MAZIERES, D., AND BONEH, D. Hacking blind. In IEEE Symposium on Security and Privacy (SP), 2014 (2014), IEEE, pp. 227–242.

[7] BLETSCH, T., JIANG, X., AND FREEH, V. Mitigating Code-reuse Attacks with Control-flow Locking. In Proceedings of the 27th Annual Computer Security Applications Conference (New York, NY, USA, 2011), ACSAC ’11, ACM, pp. 353–362.

[8] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In Proceedings of the 15th ACM Conference on Computer and Communications Security (New York, NY, USA, 2008), CCS ’08, ACM, pp. 27–38.

[9] CHIUH, T.-C., AND HSU, F.-H. RAD: a compile-time solution to buffer overflow attacks. In Proceedings 21st International Conference on Distributed Computing Systems (Apr 2001), pp. 409–417.

[10] COWAN, C., BEATTIE, S., DAY, R. F., PU, C., WAGLE, P., AND WALTHINSEN, E. Protecting systems from stack smashing attacks with stackguard. In Linux Expo (1999).

[11] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In USENIX Security Symposium (1998), vol. 98, San Antonio, TX, pp. 63–78.

[12] DANG, T. H., MANIATIS, P., AND WAGNER, D. The Performance Cost of Shadow Stacks and Stack Canaries. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (New York, NY, USA, 2015), ASIA CCS ’15, ACM, pp. 555–566.

[13] DAVI, L., SADEGHI, A.-R., AND WINANDY, M. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (New York, NY, USA, 2011), ASIACCS ’11, ACM, pp. 40–51.

[14] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOUDOSKOS, S. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (New York, NY, USA, 2015), CCS ’15, ACM, pp. 901–913.

[15] FRANTZEN, M., AND SHUEY, M. Stackghost: Hardware facilitated stack protection. In USENIX Security Symposium (2001), vol. 112.

[16] FRATRIC, I. ROPGuard: Runtime prevention of return-oriented programming attacks, 2012.

[17] G UERON, S. Intel® advanced encryption standard (aes) new instructions set. Intel Corporation (2010).

[18] HENNING, J. L. Spec cpu2000: Measuring cpu performance in the new millennium. Computer 33, 7 (2000), 28–35.

[19] HENNING, J. L. SPEC CPU2006 Benchmark Descriptions. SIGARCH Comput. Archit. News 34, 4 (Sept. 2006), 1–17.

[20] HUDN, R., WILMENS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In 2013 IEEE Symposium on Security and Privacy(SP) (05 2013), vol. 00, pp. 191–205.

[21] INTEL. Control-flow enforcement technology preview, 2016.

[22] KATOCH, V. Whitepaper on bypassing aslr/dep, 2011.

[23] KUZNETSOV, V., ZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In OSDI (2014), vol. 14, p. 00000.

[24] LATTNER, C., AND ADVEN. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the International Symposium on Code Generation and Optimization:
Feedback-directed and Runtime Optimization (Washington, DC, USA, 2004), CGO ’04, IEEE Computer Society, pp. 75–.

[25] MARCO-GISBERT, H., and RIPOLL-RIPOIL, I. Exploiting Linux and PaX ASLRs weaknesses on 32-and 64-bit systems.

[26] MASHTIZADEH, A. J., BITTATU, A., BONEH, D., and MAZIERES, D. CCFI: Cryptographically Enforced Control Flow Integrity. In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (New York, NY, USA, 2015), CCS ’15, ACM, pp. 941–951.

[27] Mcgregor, J. P., Karig, D. K., Shi, Z., and Lee, R. B. A processor architecture defense against buffer overflow attacks. In Information Technology: Research and Education, 2003. Proceedings. ITRE2003. International Conference on (2003), IEEE, pp. 243–250.

[28] Ozdoganoglu, H., Vijaykumar, T., Brodley, C. E., Kuperman, B. A., and Jalote, A. Smashguard: A hardware solution to prevent security attacks on the function return address. IEEE Transactions on Computers 55, 10 (2006), 1271–1285.

[29] Park, Y.-J., and Lee, G. Repairing return address stack for buffer overflow protection. In Proceedings of the 1st Conference on Computing Frontiers (New York, NY, USA, 2004), CF ’04, ACM, pp. 335–342.

[30] Prasad, M., and Chiueh, T.-C. A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks. In USENIX Annual Technical Conference, General Track (2003), pp. 211–224.

[31] Qualcomm Technologies, I. Pointer authentication on armv8.3, 2017.

[32] Seibt, J., Okhravi, H., and Söderström, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (New York, NY, USA, 2014), CCS ’14, ACM, pp. 54–65.

[33] Shacham, H. 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 (New York, NY, USA), CCS ’07, ACM, pp. 552–561.

[34] Shacham, H., Page, M., Paff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In Proceedings of the 11th ACM conference on Computer and communications security (2004), ACM, pp. 298–307.

[35] SiFive. SiFive’s freedom platforms, 2015.

[36] Sppardurai, S., Zhao, Q., and FaI WONG, W. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[37] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the Memory Secrecy Assumption. In Proceedings of the Second European Workshop on System Security (New York, NY, USA, 2009), EUROSEC ’09, ACM, pp. 1–8.

[38] Szekeres, L., Payer, M., Wei, T., and Song, D. SoK: Eternal War in Memory. Proceedings of the IEEE Symposium on Security and Privacy (SP) (2013), 48–62.

[39] TEAM, P. PaX address space layout randomization (ASLR).

[40] TEAM, T. C. Clang 3.8 documentation safestack, 2015.

[41] Tencent’s XuanwuLab. Exceptions in exceptions abusing special cases in system exception handling to achieve unbelievable vulnerability exploitation, 2016.

[42] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, U., Lozano, L., and Pike, G. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In USENIX Security (2014), vol. 26, pp. 27–40.

[43] UC Berkeley Architecture Research. Rocket chip generator, 2012.

[44] UC Berkeley Architecture Research. The risc-v instruction set architecture, 2015.

[45] Van der Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., and Giuffrida, C. A tough call: Mitigating advanced code-reuse attacks at the binary level. In 2016 IEEE Symposium on Security and Privacy (SP) (2016), IEEE, pp. 934–953.

[46] Wagle, P., Cowan, C., et al. Stackguard: Simple stack smash protection for gcc. In Proceedings of the GCC Developers Summit (2003), Citeseer, pp. 243–255.

[47] Weicker, R. P. Dhrystone: A synthetic systems programming benchmark. Commun. ACM 27, 10 (Oct. 1984), 1013–1030.

[48] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity and randomization for binary executables. In 2013 IEEE Symposium on Security and Privacy (SP) (2013), IEEE, pp. 559–573.
A Benchmark Size Reduction

Our evaluations takes over 30 runs of the whole SPEC 2000, but one run will take over 60 hours, so the whole evaluation will take several months. Consequently we used reduced version of reference input in some benchmarks. We will introduce how we reduced the benchmarks here.

Table 7: Benchmark Size Reduction

| Benchmark | Reduction                               | Original Runtime | Reduced Runtime |
|-----------|-----------------------------------------|------------------|-----------------|
| gzip      | Reduce buffer size from 60MB to 30MB    | 6.0h             | 3.0h            |
| vpr       | Run 1 task out of 2 tasks               | 4.3h             | 2.1h            |
| gcc       | Run 3 tasks out of 5 tasks              | 4.3h             | 2.3h            |
| mcf       | No Reduction                            | 3.1h             | 3.1h            |
| crafty    | No Reduction                            | 2.9h             | 2.9h            |
| parser    | Reduce the input from 7760 lines to 3000 lines | 5.7h | 1.7h            |
| eon       | Run 1 task out of 3 tasks               | 13.9h            | 4.0h            |
| perlbmk   | Run 4 task out of 7 tasks               | 4.5h             | 2.0h            |
| gap       | No Reduction                            | 2.1h             | 2.1h            |
| vortex    | No Reduction                            | 3.8h             | 3.8h            |
| bzip2     | Reduce buffer size from 58MB to 30MB    | 3.8h             | 1.9h            |
| twolf     | Slow 10 to 5                            | 6.5h             | 3.3h            |
| Total     |                                         | 61.1h            | 32.3h           |

For mcf, crafty, gap, vortex, the runtime is acceptable, so we did not reduce the size of them. For gzip and bzip2, we reduced the buffer size from 60MB/58MB to 30MB, the runtime is reduced by about 50%. For vpr, gcc, eon and perlbmk, we randomly chose a subset of the tasks. Parser is a syntactic parser of English, we reduced the input from 7760 lines to the first 3000 lines. The twolf is used in the process of creating the lithography artwork needed for the production of microchips. In the parameter file of twolf, the parameter "slow" will affects the number of attempts in the algorithm. We reduced the value of "slow" from 10 to 5, so the runtime reduced by around 50%.

We have reduced the runtime of some of the benchmarks, but guaranteed that the runtime will not be too short (at least around 2 hours). These reductions are completely fair and will not cause bias.