[1]

Practical Software-Based Shadow Stacks on x86-64

CHANGWEI ZOU, UNSW Sydney, Australia
YAOQING GAO, Huawei Toronto Research Center, Canada
JINGLING XUE, UNSW Sydney, Australia

Control-Flow Integrity (CFI) techniques focus often on protecting forward edges and assume that backward edges are protected by shadow stacks. However, software-based shadow stacks that can provide performance, security and compatibility are still hard to obtain, leaving an important security gap on x86-64. In this paper, we introduce a simple, efficient and effective parallel shadow stack design (based on LLVM), FlashStack, for protecting return addresses in single- and multi-threaded programs running under 64-bit Linux on x86-64, with three distinctive features. First, we introduce a novel dual-prologue approach to enable a protected function to thwart the TOCTTOU attacks, which are constructed by Microsoft's red team and lead to the deprecation of Microsoft's RFG. Second, we design a new mapping mechanism, Segment+Rsp-S, to allow the parallel shadow stack to be accessed efficiently while satisfying the constraints of `arch_prctl()` and ASLR in 64-bit Linux. Finally, we introduce a lightweight inspection mechanism, SideChannel-K, to harden FlashStack further by detecting entropy-reduction attacks efficiently and protecting the parallel shadow stack effectively with a 10-ms shuffling policy. Our evaluation on `SPEC CPU2006`, `Nginx` and `Firefox` shows that FlashStack can provide high performance, meaningful security, and reasonable compatibility for server- and client-side programs on x86-64.

ACM Reference Format:
Changwei Zou, Yaoqing Gao, and Jingling Xue. 2021. Practical Software-Based Shadow Stacks on x86-64. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2021), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As warned by Microsoft Security Response Center [1], "attackers are unanimously corrupting the stack and most Control Flow Guard (CFG) improvements will provide little value-add until stack protection loads." Even if Intel Control-Flow Enforcement Technology (CET) has been released recently [2], market penetration is estimated to take more than 6 years [1]. However, hundreds of millions of existing systems (including servers and desktop applications) will still need to be protected without the hardware support of Intel CET. Therefore, software-based mechanisms for protecting indirect control-flow edges, including both forward edges (indirect function calls and indrect jumps) and backward edges (function returns), are still needed.

In this paper, we focus on protecting backward edges. Figure 1 illustrates how indirect control-flow (forward and backward) edges are protected by several representative techniques (including Address Space Layout Randomization (ASLR) [3], re-randomization (shuffling) in code plane [4, 5], Control-Flow Integrity (CFI) [6–8], and shadow stacks [9, 10]) and how these existing techniques

[1] **New Paper, Not an Extension of a Conference Paper**

Authors' addresses: Changwei Zou, changwei.zou@unswalumni.com, UNSW Sydney, Australia; Yaoqing Gao, yaoqing.gao@huawei.com, Huawei Toronto Research Center, Canada; Jingling Xue, j.xue@unsw.edu.au, UNSW Sydney, Australia.

XXXX-XXXX/2021/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

		Our Work : TOCTTOU-Mitigated, SEGMENT+RSP-S, SIDECHANNEL-K
Layer 3	CFI	**Shadow Stack**
Layer 2	Re-Randomization (Shuffling) in Code Plane	
Layer 1	ASLR	
Protected Targets	Forward Edges	Backward Edges

Fig. 1. Past work for protecting forward and backward edges vs. this work for protecting backward edges.

are related to our work. Each of these techniques can be understood to provide protection for forward and/or backward edges at one of the three layers depicted.

At Layer 1, ASLR [3] is enabled by default on modern operating systems like Linux for protecting both forward and backward edges, by randomizing the locations of data and code sections at load time, making it more difficult for the attacker to predict certain target addresses needed. However, ASLR is known to be vulnerable to information leakage attacks [11–13].

At Layer 2, re-randomization mechanisms [4, 5], which shuffle code sections at runtime, can be deployed optionally for protecting both forward and backward edges. To support code re-randomization, however, pointer tracking can be overly time-consuming. For example, due to its excessive overheads (e.g., 35 seconds for `Nginx`), RUNTIMEASLR [4] only applies re-randomization to a child process at the time of a `fork()` call. Thus, it cannot be used to protect a program from information disclosure attacks launched after the `fork()` call. As for SHUFFLER [5], its performance overhead is as high as 30% with a shuffling period of 50 milliseconds for `Nginx`.

At Layer 3, CFI techniques [6–8] often focus on protecting forward edges and assume that backward edges are protected by shadow stacks. Although CFI can also be used to protect return addresses, the resulting attack surface is still too large [14, 15]. Since a function may be called directly or indirectly in many different call sites, a static CFI approach for protecting its function return will often result in a coarse-gained solution, leaving still enough leeway for the attacker. For example, the return of `printf()` in FreeBSD has over 5,000 permitted targets [16]. Thus, fully-precise shadow stacks are often recommended to be used for protecting return addresses.

Problem Statement. In this paper, we focus on developing a software-based shallow stack design for protecting return addresses. Currently, software-based shadow stacks that can provide performance, security and compatibility for both server- and client-side programs on x86-64 are still hard to obtain [1, 17]. Inspired by RFG (Return Flow Guard) [1] developed for Windows by Microsoft but deprecated now, we introduce a practical software-based solution for 64-bit Linux.

Broadly speaking, there are two types of shadow stacks: parallel [9, 10] and compact [10, 18] shadow stacks. A parallel shadow stack has the same stack size as its call stack and can thus reuse the call stack pointer `rsp` to access the shadow stack. As a result, a parallel shadow stack does not need to handle stack unwinding due to setjmp/longjmp and C++ exceptions [10]. In contrast, a compact shadow stack has a smaller stack size than its call stack but must maintain more expensively a separate shadow stack pointer to access the shadow stack, thereby incurring also some additional performance overheads in handling stack unwinding [10].

For x86-64, the cruel reality is that shadow stacks are not widely deployed for two reasons. First, whether parallel or compact shadow stacks should be used is still being investigated. Our recent design (named BuddyStack) adopts parallel shadow stacks [19], as in Microsoft's RFG (on Windows), to trade memory for performance and compatibility. However, some other designs [10, 17] make the opposite tradeoff by adopting compact shadow stacks, thereby suffering from either high performance overheads or severe compatibility problems. For example, LLVM's ShadowCallStack has been deprecated due to its performance and security deficiencies [17], while Shadesmar [10] trades compatibility for performance by reserving a general register (e.g., `r15`) to maintain the shadow stack pointer (but it often fails to build or run large projects like `Firefox`). Second, the well-known Time-Of-Check To Time-Of-Use (TOCTTOU) attacks [1] as demonstrated by Microsoft's red team (in Windows) have undermined the effectiveness of all existing shadow stack designs on x86-64, leading to the deprecation of Microsoft's RFG (on Windows).

As long as this important security gap remains to exist, attackers need not bother to turn to more advanced and complex data-only attacks [20, 21], since the return addresses on the call stack are usually substantially easier to corrupt.

To protect a shadow stack in BuddyStack [19], we re-randomize its return addresses by mutating a random value added to each return address efficiently. In this paper, we will continue to focus on protecting return addresses with a different shadow stack design. This time, however, we aim to provide stronger protection by shuffling a shadow stack in user space to make it fast-moving continuously instead, at a shuffling overhead of 0.13% only (Section 6.1).

Design Choices. When designing a shadow stack mechanism for x86-64, two design choices must be reconsidered carefully. First, should we adopt a parallel or compact shadow stack design to protect multi-threaded server programs and browsers, the most popular victims in cyberattacks, while keeping compatibility problems to a minimum? Second, which register should we reserve in order to keep track of the shadow stack associated with the call stack in each thread?

In this paper, we have adopted a parallel shadow stack design as in Microsoft's RFG [1] (developed for Windows). For 64-bit Linux, a recent evaluation that we conducted during the development of BuddyStack [19] shows that a parallel shadow stack design does not lead to high memory overheads on x86-64, since a call stack usually consumes tens of KBs of physical memory in real-world applications. To keep track of a fast-moving shadow stack in our design, we need to make use of some register. We have not settled with a general register as in Shadesmar [10] due to its severe compatibility issues with unprotected libraries (caused by register clashes) [19]. Therefore, we are left with selecting one from the two legacy segment registers on x86-64, `fs` and `gs`. We have opted to use the latter since the former has been used in implementing thread-local storage (TLS) in the standard thread library.

Challenges. With these two design choices considered, there are still several challenges faced in protecting return addresses. First, how do we mitigate the TOCTTOU attacks [1, 22], which have actually caused Microsoft's RFG to be deprecated, and consequently, challenged the effectiveness of all other shadow stack designs on x86-64? Second, how can we adapt a scheme used in Microsoft's RFG, denoted Segment+Rsp in this paper, which uses a segment register (i.e., `fs`) and the call stack pointer Rsp for accessing the parallel shadow stack in Windows, to work also in 64-bit Linux by maintaining its efficiency while also satisfying the constraints of `arch_prctl()`(a system call for setting architecture-specific process or thread state) [23, 24] and ASLR in 64-bit Linux? Third, how do we efficiently and effectively mitigate existing information disclosure and side channel attacks such as thread spraying, memory oracles and cache attacks [12, 13, 25] (used to expose hidden shadow stacks) in our shadow stack design?

Our Approach. In this paper, we introduce a practical parallel shadow stack design, FlashStack, for protecting return addresses in single- and multi-threaded programs running under 64-bit Linux on x86-64. FlashStack can provide high performance, meaningful security, and reasonable compatibility due to its three distinctive features.

```
01 void VulnerableThread(void *arg){
02   while(1){
03     *ret_slot = shell_code_addr;
04   }
05 }

06 void GetLength(){
07   …
08   // StrnLen is a leaf function
09   StrnLen();
10   …
11 }

12 void WorkerThread(void *arg){
13   while(!stop){
14     Sleep();
15     GetLength();
16   }
17 }
```

Fig. 2. A TOCTTOU attack on x86-64 constructed in [1].

First, FlashStack is designed to provide stronger protection against the TOCTTOU attacks [1], which have undermined the effectiveness of all existing shadow stack designs on x86-64. The TOCTTOU vulnerability on x86-64 represents a troublesome obstacle for deploying shadow stacks. As a call instruction on x86-64 causes the return address to be pushed to the call stack rather than saved in a register as on ARM, there is a TOCTTOU gap on x86-64 [1, 10]. Specifically, the pushed return address may be corrupted by one thread before it is fetched by another thread in a multi-threaded program. As shown in Figure 2, the TOCTTOU attack constructed by Microsoft's red team [1] works by exploiting a locally-running vulnerable thread (lines 1-5) to write constantly into the return slot of a leaf function (line 9) that is repeatedly called in the attacked thread (lines 12-17). Due to such TOCTTOU attacks, the effectiveness of Microsoft's RFG [1] is challenged. To mitigate the TOCTTOU attacks, we apply a novel dual-prologue approach for a protected function (one before the call instruction and one at the beginning of the function) to eliminate completely the TOCTTOU attacks for instrumented call instructions while providing also compatibility with non-instrumented call instructions in unprotected code (for incremental deployment [26, 27]).

Second, FlashStack extends Segment+Rsp (used in Microsoft's RFG) into Segment+Rsp-S in order to access the parallel shadow stack efficiently in 64-bit Linux while satisfying the constraints of `arch_prctl()` and ASLR, where Segment represents a segment register (i.e., `gs`), Rsp is the call stack pointer, and S symbolizes the size of the whole user space. Note that only the visible part (the 16-bit segment selector) of gs is saved in `ucontext_t` (a structure type suitable for holding the context for a user thread of execution) [28], rather than its internal 64-bit base address (of the parallel shadow stack). Thus, the shadow stack cannot be exposed by only peeking the spilled gs (say, due to signals) in user space.

Finally, FlashStack exploits side channel information during shuffling by applying SideChannel-K, a lightweight inspection mechanism driven by a 10-ms shuffling policy, to enforce stack integrity, where K is the number of memory allocation attempts needed to allocate a new shadow

stack randomly. Existing information disclosure and side channel attacks such as thread spraying, memory oracles and cache attacks [12, 13, 25] consume abnormally a large amount of virtual memory space ($\gg$ 1 TB) to expose a hidden safe region. For example, AnC [25] needs to allocate a 4TB virtual memory region before launching its side channel cache attacks. However, as reported in [29] and confirmed in our evaluation (Section 6.2), programs rarely consume terabytes of memory. By leveraging the side channel information, SideChannel-K can detect efficiently these entropy-reduction attacks [12, 13, 25] and avoid greatly unnecessary inspections (Section 5.3). In addition, as out-of-bounds accesses are usually rare, by slowing down the malicious crash-resistant probing to be no more than 20 faults per second, about 6,990 minutes will be needed for CROP attacks [30] to expose a fixed position shadow stack (say, of a maliciously-blocked thread) in browsers, much longer than the average daily time (145 minutes per day) spent by internet users worldwide [31]. As for servers (e.g., Nginx), which are reported to often sit tens of milliseconds away from attacker machines [5], our 10-ms shuffling policy is fast enough to make obsolete the possibly-leaked location of a shadow stack, at the expense of only about 13 μs per randomization.

This paper makes the following contributions:

- The first shadow stack design for thwarting the TOCTTOU attacks, which undermine the effectiveness of all existing software-based shadow stacks on x86-64.
- Segment+Rsp-S, a new mapping scheme, which can maximize performance when accessing a parallel shadow stack (by adding at most three additional memory accesses) in 64-bit Linux.
- SideChannel-K, a lightweight inspection mechanism driven by a 10-ms shuffling policy, which can mitigate information disclosure and side channel attacks from locally-running scripts, by exploiting the side channel information during shuffling.

2 BACKGROUND AND MOTIVATION

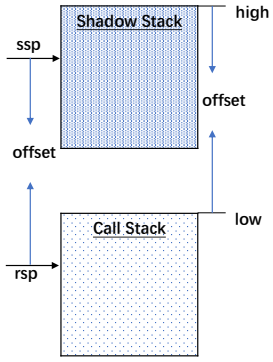
In Section 2.1, we explain why the Segment+Rsp scheme adopted in Microsoft's RFG on Windows for accessing the parallel shadow stack does not work directly in 64-bit Linux and then motivate a need for its adaptation (into our new scheme, Segment+Rsp-S). In Section 2.2, we highlight the advantages of FlashStack by comparing it with three different shadow stack designs in terms of their instrumentation overheads.

2.1 Why Segment+Rsp Does Not Work in 64-bit Linux?

Figure 3 gives an analogue of Microsoft's RFG for 64-bit Linux, using `gs` rather than `fs`. By setting the `gs_base` (line 1) to be the same as the offset from the shadow stack to its call stack, the shadow stack pointer `ssp` can be found quickly:

$$\texttt{ssp} = \texttt{rsp} + \texttt{offset} \tag{1}$$

The shadow stack can then be accessed efficiently in the form of `%gs:(%rsp)` (lines 7 and 10). However, `offset`, the second argument in `arch_prctl()` (line 2), cannot be negative. This implies that the shadow stack must be higher than the call stack in user space, which is not feasible under Linux, as the call stack (with 40-bit entropy [13]) is at the top of user space. Due to ASLR [3], the




```
01 // gs_base = offset
02 // 0 <= offset <= 0x7FFFFFFFEFFF
03 arch_prctl(ARCH_SET_GS, offset);

04  # prologue
05  mov (%rsp), %r11
06  # ssp = gs_base + rsp
07  mov %r11, %gs:(%rsp)
    ...
08  # epilogue
09  # ssp = gs_base + rsp
10  mov %gs:(%rsp), %r11
11  cmp (%rsp), %r11
12  jne fastfail
13  ret
14 fastfail:
15  ud2
```

Fig. 3. Microsoft's RFG rewritten for 64-bit Linux.

```
01 # Prologue
02 mov    (%rsp), %rax
03 mov    %rax, (%r15)
04 mov    %rsp, 0x8(%r15)
05 lea    0x10(%r15), %r15

06 # Epilogue
07 mov    -0x10(%r15), %r11
08 lea    -0x10(%r15), %r15
09 pop    %r10
10 xor    %r10, %r11
11 popcnt %r11, %r11
12 mov    0x40000fff(,%r11,1), %r11b
13 jmpq   *%r10
```

(a) SHADESMAR

```
14 # Prologue
15 mov    (%rsp), %r11
16 mov    %r11, -0x800000(%rsp)

 ....

17 # Epilogue
18 mov    -0x800000(%rsp),%r11
19 add    $0x8, %rsp
20 jmpq   *%r11
```

(b) Constant offsets

Fig. 4. SHADESMAR [10] and parallel shadow stacks with constant offsets [9].

Table 1. Instrumentation overheads measured in terms of #MEM (number of memory accesses) / #INS (number of instructions) for three existing shadow stack designs and FLASHSTACK.

	Compact (#MEM / #INS)	Parallel (#MEM / #INS)		
	SHADESMAR	Constant	RFG	FLASHSTACK
Prologue	3/4	2/2	2/2	2/3
Epilogue	3/7	1/3	3/4	1/3
Total	6/11	3/5	5/6	3/6

call stack may randomly appear near the highest address in user space, so that there may not be any space left for its shadow stack at a higher address.

In this paper, we introduce SEGMENT+RSP-S for 64-bit Linux, which extends SEGMENT+RSP by relying on a novel mapping mechanism proposed in this paper (Section 4).

2.2 Instrumentation Overheads

Table 1 highlights the advantages of FLASHSTACK (our shadow stack design) by comparing it with three popular shadow stack designs (illustrated in Figures 3 and 4) in terms of their instrumentation overheads incurred. The instructions that need to access memory are underlined.

SHADESMAR [10], which uses a compact shadow stack, trades compatibility for performance, by using 4 instructions (3 memory accesses) at the prologue and 7 instructions (3 memory accesses) at the epilogue (Figure 4(a)).

A constant-offset-based parallel shadow stack (Figure 4(b))) is the most efficient [9], requiring the smallest number of memory accesses (two at the prologue and one at the epilogue). However, such a traditional parallel stack is not movable, trading security for performance (in addition to suffering from compatibility problems with multi-threading [10]). In this paper, we propose a new gs-based parallel shadow stack, FlashStack, that is fast-moving in user space, by retaining the performance benefits of a non-movable parallel shadow (requiring at most three memory accesses at the prologue and epilogue of a protected function) while also improving significantly its security and compatibility deficiencies.

By using two prologues for a protected function (one at its beginning and one at its corresponding call instruction) in a novel way, we can not only provide stronger protection against the TOCTTOU attacks but also lower the number of memory accesses at the prologue for a direct call from two to one. Note that fewer memory accesses and instructions do lead to better performance, but not proportional (Section 6.1).

3 THREAT MODEL

In this paper, we assume an adversary with arbitrary read and write primitives. This threat model is the same as those adopted in previous research on defending against control-flow hijacking, such as CFI [32, 33] and shadow stacks [9, 10]. The aim of the adversary is to launch control-flow hijacking attacks by exploiting existing vulnerabilities and corrupting return addresses. Protection for function and virtual table pointers can be provided orthogonally by defenses such as CFI [32, 33]. Data-only attacks [20, 21] are out of scope for this paper.

We also assume that standard defenses such as ASLR [3] and DEP [34] have already been deployed to further constrain the adversary. We focus on protecting applications in user space. How to protect the OS kernel [35–37] is also out of scope of this paper.

4 METHODOLOGY

FlashStack consists of (1) an efficient mapping mechanism from the call stack pointer `rsp` to the shadow stack pointer `ssp`, (2) a lightweight inspection mechanism driven by a 10-ms shuffling policy, (3) a protected thread-local storage mechanism for storing security-critical metadata, and (4) a two-pronged strategy for using two prologues to instrument a protected function to mitigate the TOCTTOU attacks.

4.1 Shadow Stack Mapping

To ensure that the offset from the shadow stack to the call stack is always positive under 64-bit Linux, we first transform the shadow stack to a virtual shadow stack outside of user space and then apply the inverse transformation to return to user space. In Figure 5, we illustrate the mapping used in FlashStack to access efficiently the shadow stack from the call stack pointer `rsp`. In the 47-bit user space, as the call stack is randomly placed at the higher side with 40-bit entropy [13] in Linux, FlashStack will always allocate its shadow stack below, yielding a negative offset between the two. To "interpret" such a negative offset as a positive offset, we assume that there is a virtual (i.e., imaginary) shadow stack above user space, whose top address `v_top` is the sum of the top address `s_top` of the real shadow stack and the size of user space `S` (i.e., 0x800000000000), `v_top = s_top + S`, as shown at the first step ❶. Thus, the virtual shadow stack is always higher than the call stack, so that we can always obtain a positive offset between the virtual shadow stack and the call stack, `offset = v_top - c_top`, where `c_top` is the top address of the call stack, at the second step ❷.

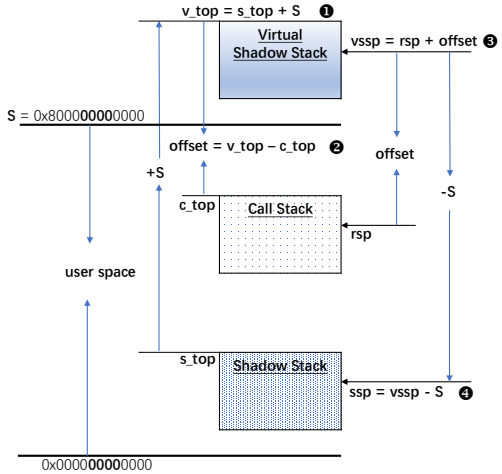


Fig. 5. FlashStack's Segment+Rsp-S for accessing the parallel shadow stack under 64-bit Linux.

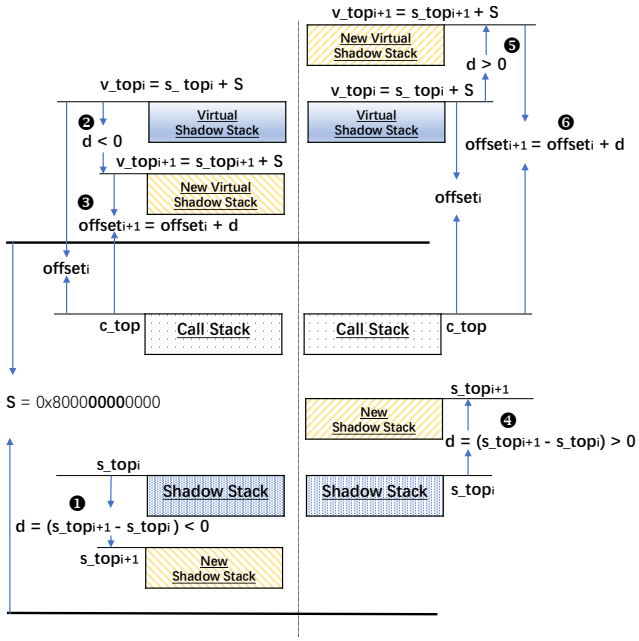


Fig. 6. FlashStack's mechanism for shuffling the shadow stack continuously in 64-bit Linux.

By combining these two steps together, we can now use the following offset as the `gs` base in `arch_prctl(ARCH_SET_GS, offset)`:

$$\texttt{offset} = \texttt{s\_top} + \texttt{S} - \texttt{c\_top} \tag{2}$$

The virtual shadow stack pointer `vssp` can now be found easily as `vssp = rsp + offset` at the third step ❸. Finally, we can apply the inverse of the transformation performed at the first step to obtain the real shadow stack pointer `ssp` as `ssp = vssp - S` at the fourth step ❹.

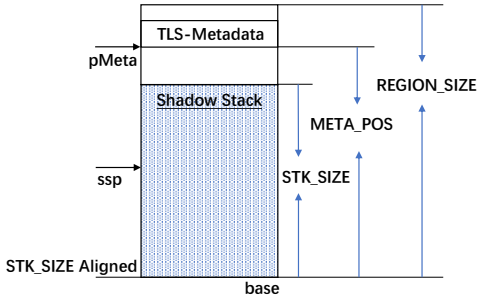


Fig. 7. FlashStack's protected thread-local storage.

Similarly, by combining the last two steps, we obtain a novel mapping mechanism, Segment+Rsp-S, for mapping the call stack pointer `rsp` to the real shadow stack pointer `ssp`:

$$\texttt{ssp} = \texttt{offset} + \texttt{rsp} - \texttt{S} \tag{3}$$

4.2 Fast-Moving Shadow Stack

When protecting return addresses with a parallel shadow stack, we must enforce its integrity. In Figure 6, we illustrate a shuffling mechanism used in FlashStack to make the shadow stack continuously fast-moving, so that information disclosure attacks [12, 13] can be mitigated.

The shadow stack can be remapped to a new position by using the system call `mremap()` in user space. Figure 6 illustrates the shuffling process from time `i` to time `i+1`. There are two cases to analyze, depending on whether the new shadow stack (created at time `i+1`) is below the old shadow stack (created at time `i`) or above.

We first discuss the former case, accomplished in three steps ❶ - ❸. At the first step ❶, the distance $\texttt{d} = \texttt{s\_top}_{\texttt{i+1}} - \texttt{s\_top}_{\texttt{i}}$ between the new shadow stack and the old one is negative, where $\texttt{s\_top}_{\texttt{i+1}}$ and $\texttt{s\_top}_{\texttt{i}}$ are the top addresses of the new and old shadow stacks, respectively. By construction, `d` also represents exactly the same distance between the two corresponding virtual shadow stacks (above user space) at ❷. As the shadow stack is shuffled from time `i` to time `i+1`, the corresponding virtual shadow stack is also shuffled in exactly the same way (by the same amount relative to the call stack). Thus, the offset between the call stack and the virtual shadow stack can be simply adjusted at ❸ as follows:

$$\begin{aligned} \texttt{offset}_{\texttt{i+1}} &= \texttt{offset}_{\texttt{i}} + \texttt{d} \\ &= \texttt{offset}_{\texttt{i}} + \texttt{s\_top}_{\texttt{i+1}} - \texttt{s\_top}_{\texttt{i}} \end{aligned} \tag{4}$$

where $\texttt{offset}_{\texttt{i+1}}$ ($\texttt{offset}_{\texttt{i}}$) is the new (old) offset.

Similarly, the latter case (in the right part of Figure 6) can also be analyzed in ❹ – ❻. The new shadow stack has a higher address than the old one, but we obtain exactly Equation 4 as before at ❻ even if `d` (at ❹ and ❺) is now positive.

As a new shadow stack is randomly placed in user space, it is likely that the random position generated has already been occupied. Generally speaking, with more virtual memory consumed, more attempts will be needed to hit an unmapped memory region at random. So, we can use the number of attempts as side channel information to decide whether we need to detect entropy-reduction attacks during shuffling, thus obtaining a lightweight inspection mechanism (Section 5.3).

4.3 Protected Thread-Local Storage

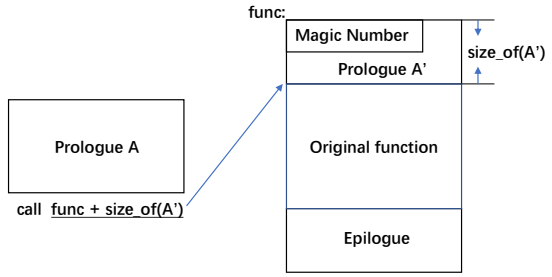


Fig. 8. FlashStack's dual prologues for mitigating the TOCTTOU attacks.

In Figure 7, we illustrate how FlashStack accesses the per-thread metadata in its thread-local storage (TLS), based on the shadow stack mapping mechanism given in Figure 5, such that the security-critical metadata stored in our TLS can also be protected during shuffling.

In FlashStack, the parallel shadow stack itself is allocated at a memory position aligned on `STK_SIZE` (say, 8MB). The shadow stack pointer (`ssp`) can be calculated from the call stack pointer `rsp` according to Equation 3. As illustrated in Figure 7, the location of the per-thread metadata (pointed by `pMeta`) can be found at:

$$\texttt{pMeta} = (\texttt{ssp}\ \&\ -\texttt{STK\_SIZE}) + \texttt{META\_POS} \tag{5}$$

where `META_POS` is the offset of `TLS-Metadata` to the base address (`(ssp & -STK_SIZE)`) as shown in Figure 7. This base address can be calculated efficiently by using a bit-and operation between `ssp` and `-STK_SIZE`. Apparently, `META_POS` should be larger than `STK_SIZE`.

Unlike existing TLS mechanisms [38], our TLS is also shuffled (protected) with its corresponding shadow stack, providing better security guarantees for security-critical metadata.

4.4 Dual Prologues

In Figure 8, we illustrate a novel two-pronged instrumentation technique used in FlashStack for mitigating the TOCTTOU attacks on x86-64. We use two prologues for a protected function. In addition to the normal prologue `A'` instrumented at the beginning of the protected function, we add also another prologue `A` before a call instruction. The return address of the call instruction can be calculated at `A` and directly moved to the shadow stack, so that the normal prologue `A'` can be skipped at the call instruction by incrementing the target function address by `size_of(A')` bytes.

However, we must still retain the normal prologue `A'` for two reasons. First, its first eight bytes can be used as a signature (i.e., a magic number) for checking whether a target function is protected (when instrumenting an indirect call). Second, the protected function may be invoked as a callback in unprotected code, where its call instructions are not instrumented. This ensures that FlashStack is compatible with unprotected code, as incremental deployment is essential for a hardening tool to be adopted [26, 27].

5 DESIGN AND IMPLEMENTATION

We describe the design and implementation of FlashStack, as shown in Figure 9. FlashStack uses LLVM for translating source code into assembly code and AFL [39, 40] to produce instrumented assembly code. To provide compatibility with a range of LLVM backends, FlashStack is designed as an assembly instrumentator based on AFL. In addition, we would also like to have a design that can be extended to cover the hand-written assembly files in large projects [19]. The assembly code generated will be intercepted by our assembly instrumentator to instrument call instructions and functions (Section 5.1), and then passed to the native assembler and linker to generate a hardened

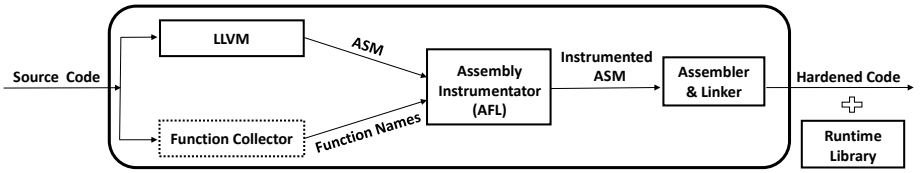


Fig. 9. The workflow of FlashStack.

binary. FlashStack's fast-moving shadow stack mechanism is supported by our runtime library (Section 5.3). The function collector is designed as a preprocessing pass to collect the names of all protected functions, which are later used in instrumenting a direct call (Section 5.1). FlashStack is designed as a drop-in replacement for LLVM's own experimental shadow stack implementation (ShadowCallStack [17]) to facilitate its adoption.

5.1 Assembly Code Instrumentation

```
01 foo:
02 # Prologue
03 400850: 4c 8b 1c 24  movq (%rsp),%r11
04 400854: 49 ba 00 00  movq $-0x800000000000,%r10
           00 00 00 80
           ff ff
05 40085e: 65 4e 89 1c  movq %r11,%gs:(%rsp,%r10,1)
           14
    ...
06 # Epilogue
07 4008a1: 49 ba 00 00  movq $-0x800000000000,%r10
           00 00 00 80
           ff ff
08 4008ab: 48 83 c4 08  addq $0x8,%rsp
09 4008af: 65 42 ff 64  jmpq *%gs:-0x8(%rsp,%r10,1)
           14 f8
```

Fig. 10. Assembly instrumentation at the prologue and epilogue of a protected function `foo()` on x86-64.

Figure 10 gives the prologue and epilogue of a protected function `foo()` used. There are two memory accesses at the prologue and one at the epilogue, as underlined. The `offset` in Equation 2 is set in our runtime library by invoking `arch_prctl()` when a thread starts. Equation 3 is implemented in lines 4 and 5 at the prologue and lines 7 and 9 at the epilogue in Figure 10. The immediate number `0x800000000000` (lines 4 and 7) is the size of user space in 64-bit Linux (i.e., S in Figure 5). As S is larger than a 32-bit int, it cannot be coded as a 32-bit displacement in the assembly instructions in lines 5 and 9. Thus, we have stored it in register `r10` (lines 4 and 7).

In line 9, preferring `jmp` to `ret` is a reasonable trade-off between performance and security for several reasons. First, this closes the TOCTTOU window at the epilogue [1]. Second, we can mitigate the attacks that exploit the speculative execution of return stack buffer [41]. Finally, omitting rets does not really hurt FlashStack's performance (Table 6). Return stack buffer [41] aims to speed up the return on the call stack only. So the required memory accesses on the shadow stack must be made no matter whether rets are used or not.

Like BuddyStack [19], FlashStack does not validate a return address on the shadow stack against the corresponding return address on the call stack at the epilogue of a protected function, since such a validation will be meaningless once the shadow stack has been exposed. In other words, the attacker should have already corrupted the two return addresses at the same time (in one payload) with arbitrary writes [42] if they know where the hidden stack stack is.

```
01 # direct call
02 #   callq     foo
03 # modified as follows
04 # save the return address to r11
05      leaq      1f(%rip), %r11
06      movq      $-0x800000000000, %r10
07 # move r11 to shadow stack
08      movq      %r11, %gs:-8(%rsp, %r10, 1)
09 # skip the prologue
10      callq     foo+19
11  1:
12      ...
```

(a) A direct call

```
13 # indirect call
14 #   callq     *%rax
15 # modified as follows
16      movq      $0xba49241c8b4c, %r11
17      cmpq      (%rax), %r11
18      jne       1f
19      leaq      2f(%rip), %r11
20      movq      $-0x800000000000, %r10
21      movq      %r11, %gs:-8(%rsp, %r10, 1)
22      addq      $0x13, %rax
23  1:
24      callq     *%rax
25  2:
26      ...
```

(b) An indirect call

Fig. 11. FlashStack's instrumentation at direct and indirect calls on x86-64.

The size of the prologue (i.e., `size_of(A')` in Figure 8) is 19 bytes. The first eight bytes (`0xba49241c8b4c` in little-endian order) can be used as a magic number to identify a protected function.

Now, we discuss how to instrument a call instruction in a protected function in order to mitigate the TOCTTOU attacks at the call instruction. Figure 11 shows how direct and indirect calls are instrumented by using different prologues.

For a direct call to a protected function (say, `foo()` in line 2), as shown in Figure 11(a), we add another prologue (lines 5-8) before the call instruction. By adding a label "1:" (in line 11) after the call instruction, we can use an `leaq` instruction to calculate the return address in line 5 and store it in register r11. It is worth noting that there is not any stack access in this instruction. So the number of memory accesses is reduced to only one (line 8) at this prologue. The return address is then moved directly to the shadow stack in line 8. Later, based on the function names collected by our function collector (Figure 9), we can modify the target address as `foo+19` to skip the prologue at the beginning of `foo()`, where 19 is the size of the function prologue instrumented (Figure 10).

Similarly, for an indirect call (line 14), we also add another prologue (lines 16-23) before the call instruction, as shown in Figure 11(b). The major difference here is that we need to use the first eight bytes of the target function to test whether it is a protected function in lines 16-18. The immediate number `0xba49241c8b4c` in line 16 is the integer value of the first eight bytes of a protected function (Figure 10). If it is not a protected function, then the control flow is diverted to line 23. Otherwise, lines 19-22 will be executed to move the return address (marked by the label "2:" in line 25) to the shadow stack and the normal prologue of the protected function will be skipped in line 22. The test in line 17 is necessary as FlashStack needs to be compatible with unprotected code. Our dual-prologue approach works for recursion directly. Tail-recursive calls that use jump instructions instead of call instructions can be handled similarly as shown in Figure 11. Such jump instructions can be marked by the "# TAILCALL" comments added by the LLVM compiler.

5.2 Policies for Mitigating Attacks

We explain how FlashStack mitigates the information disclosure and probing attacks against shadow stacks and their counterparts proposed in prior research [12, 13, 25, 30].

Thread spraying and allocation oracles [12, 13] can consume the virtual memory space and reduce the entropy of the attacked process. Modern browsers, such as `Chrome` and `Firefox`, have a multi-process architecture with each process being multi-threaded. A browser user may leave a large number of tabs open, increasing the number of processes and/or threads created. Suppose

that the call stack of a thread occupies 8-MB virtual memory. A browser user needs to open about 128 tabs (threads) in order to cause the underlying call stacks to consume 1-GB virtual memory, which represents only just a small percentage of the 128-TB user space provided in a single process. However, in the extreme cases, certain abnormal behaviors (e.g., launching hundreds of thousands of worker threads) can still be detected as a kind of thread spraying attack. As discussed in Section 5.3, FlashStack's inspection mechanism, which is called upon whenever a new shadow stack is allocated (Figure 13), can be used to detect such abnormal behaviors.

Server-Side Programs. The network latency is reported to be often at tens of milliseconds [5]. We can use a 10-ms shuffling policy to relocate the shadow stack for each thread continuously in user space, with each shuffle being triggered in a hooked I/O function (say, `recv()`) if this has not been done in the past 10 milliseconds.

Client-Side Programs. Side channel attacks such as AnC [25] can use an EVICT+TIME cache attack to expose hidden code and heap pointers of major browsers in 150 seconds. However, AnC [25] can only achieve this by allocating an abnormally large virtual memory region (4 TB), which can be detected by FlashStack as abnormal behaviors.

As for crash-resistant attacks [30], our evaluation shows that for a major application like `Firefox`, only several segment faults are observed during its normal execution. Tens of thousands of segment faults within a second are certainly an abnormal behavior. By instrumenting the signal handlers of `Firefox` to monitor the number of segment faults per second, FlashStack can mitigate crash-resistant attacks. The number of segment faults can be stored in a 4096-byte page (aligned on page boundaries) so that the entire page can be set as read-only after it has been updated [8].

Cross-Thread Attacks. Although having a dedicated shuffling thread may be more secure, frequent synchronizations among the threads can hurt performance. Thus, FlashStack chooses to let every thread do its own shuffling. Smart adversaries may maliciously block a thread (say, by corrupting a lock), such that its shadow stack will not be shuffled. However, even with powerful crash-resistant attacks [30], they still need to spend about 6,990 minutes to expose a fixed position shadow stack in a FlashStack-protected browser, obviously exceeding the average daily time (145 minutes per day) spent by internet users worldwide [31] (Section 6.2). In addition, by making the shadow stack of a blocked thread read-only, such cross-thread attacks can be further mitigated. Note that no crash-resistant attacks exist in server programs [30].

5.3 Runtime Library

In Figure 12, we give our implementation of FlashStack's fast-moving shadow stack mechanism in 64-bit Linux, with `shuffle_shadow_stack()` being triggered by hooking the library functions, which include `write()`, `sendto()`, `sendmsg()`, `send()`, `read()`, `recvfrom()`, `recvmsg()` and `recv()` in our current implementation (LD_PRELOAD).

In lines 2-7, `shuffle_shadow_stack()` relies on Equation 5 to access the thread-local metadata (Figure 7). Currently, FlashStack's protected TLS is used to store the time when the latest shuffling happened and a flag to mark whether a thread is being shuffled (omitted for brevity). These per-thread data will be used in line 9 to check whether randomization has been done or not in the past 10 milliseconds. In lines 10-20, `shuffle_shadow_stack()` relies on Equation 4 to move the shadow tack dynamically to mitigate information disclosure attacks. A new shadow stack is allocated in lines 10-11 by calling `get_memory_at_random()`, which is explained shortly. The old shadow stack is then remapped to the new shadow stack in lines 14-18 by calling `mremap()`. It is worth noting that the memory region occupied by the old shadow stack becomes unmapped after the call `mremap()` has returned. Afterwards, any attempt by an attacker for probing the unmapped old shadow stack will trigger a segment fault. Note that the offset (from the new virtual shadow stack to the call

```
01 void shuffle_shadow_stack(){
02   long offset = 0;
03   arch_prctl(ARCH_GET_GS, &offset);
04   long ssp = GET_RSP() + offset - USER_SPACE_SIZE;
05   long old_stack = ssp & -STK_SIZE;
06   struct metadata *pMeta =
07          (struct metadata *) (old_stack + META_POS);
08   // sanity checks omitted
09   SANITY_CHECK(pMeta, old_stack, offset);
10   void *new_stack =
11         (long *) get_memory_at_random(REGION_SIZE);
12   offset += ((long)new_stack) - old_stack;
13   // fast moving (remapping)
14   mremap((void *) old_stack,
15          REGION_SIZE,
16          REGION_SIZE,
17          MREMAP_MAYMOVE | MREMAP_FIXED,
18          new_stack);
19   // reset the offset
20   arch_prctl(ARCH_SET_GS, offset);
21 }
```

Fig. 12. Shuffling of FlashStack's shadow stack and TLS.

stack) is calculated in line 12 according to Equation 4 and then reset in line 20. Under "-O3", the local variables in Figure 12 are expected to be in registers, not the regular stack.

We show how `get_memory_at_random()` (Figure 13) allocates a new shadow stack at a random position below the call stack. First, we obtain the call stack pointer `rsp` in line 6 and calculate the largest address that can be used for a shadow stack in line 7. We then attempt to allocate a new shadow stack repeatedly by calling `mmap()` in lines 16-18 within the loop (lines 8-26) for a maximum of `MAX_MMAP_ATTEMPTS` (defaulted to 64 currently) times. In the `mmap()` call, we do not use `MAP_FIXED` since setting this flag will make `get_memory_at_random()` both less portable [43] and slightly inefficient. As illustrated in Figure 7, the shadow stack is allocated on a STK_SIZE-aligned memory position (say, 8MB on 64-bit Ubuntu). The sanity check in lines 21-24 is performed to test whether the allocated virtual memory region is indeed aligned as expected.

SideChannel-K. In practice, a real-world application rarely consumes terabytes of memory and even big data applications only use gigabytes of virtual memory space [29]. So the memory allocation call to `mmap()` in lines 16-18 can often succeed at the first attempt in the large user space (say, of size 128 TB). Naturally, we can take advantage of the number of attempts (recorded in `k` in line 28) as side channel information to check whether a protected application is undergoing entropy-reduction attacks such as thread spraying [12] and allocation oracles [13]. The test in line 28 acts as a simple and effective filter to avoid unnecessary inspections. Only when `k` is larger than a predefined `THRESHOLD` (say, 6 used in our evaluation) do we need to further check the total size of mapped memory regions in line 29. If it is larger than a reasonable size predefined (say, 1 TB), we can raise the alarm for the attack and stop the program. Section 6.2 discusses how to select a particular value of `THRESHOLD` analytically.

6 EVALUATION

Our evaluation aims to show that FlashStack offers performance, meaningful security, and reasonable compatibility for both server- and client-side programs. We demonstrate this by comparing FlashStack with Shadesmar, a state-of-the-art compact shadow stack design in building and running `SPEC CPU2006`, `Nginx` (a widely-adopted web server) and `Firefox` (a widely-used browser). Shadesmar's most efficient register-based (i.e., `r15`) scheme is used, without using any excessively

```
01 //long A[MAX_MMAP_ATTEMPTS+2];
02
03 void *get_memory_at_random(size_t size){
04   void *addr =  MAP_FAILED;
05   unsigned long k = 0;
06   unsigned long rsp = GET_RSP();
07   unsigned long max = rsp - 2 * REGION_SIZE;
08   while( MAP_FAILED == addr ){
09     unsigned long x = GEN_RANDOM_VAL();
10     x %= max;
11     x &= SHADOW_STACK_ADDR_MASK;
12     k++;
13     if(k > MAX_MMAP_ATTEMPTS){
14        break;
15     }
16     addr = mmap( (void *) x, size,
17                  PROT_READ | PROT_WRITE,
18                  MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
19     if(addr != MAP_FAILED){
20       unsigned long allocated = (unsigned long) addr;
21       if(is_aligned(allocated) || (allocated > max)){
22         munmap(addr, size);
23         addr = MAP_FAILED;
24       }
25     }
26   }
27   //__sync_fetch_and_add(&A[k], 1);
28   if(k > THRESHOLD){
29     CHECK_MAPPED_REGIONS(); // /proc/pid/statm
30   }
31   return addr;
32 }
```

Fig. 13. Allocation of FlashStack's shadow stack and TLS.

expensive protection mechanism, MPX [10] or MPK [44], for integrity protection. In addition, we will also discuss the differences between FlashStack and BuddyStack [19].

Our recent research [19] shows that the memory overhead incurred by a parallel shadow stack is quite small, costing only tens of KBs on average, in real-world single- and multi-threaded applications. As FlashStack is also a parallel shadow stack, the stack usage information reported earlier in real-world programs remains to be relevant but elided here in this paper.

Our platform consists of a 3.20 GHz Intel i5-6500 CPU (4-core) with 16 GB memory, running the 64-bit Ubuntu 18.04. For a fair comparison, both hardening tools use the the same compiler framework (LLVM 7.0). The optimization flag "-O3" is used for building `SPEC CPU2006` and `Nginx1.18` while the default build setting for `Firefox79.0` ("-O2" and "-O3" in different modules) is used.

6.1 Performance

We show that FlashStack is highly performant in running `Nginx` and `Firefox` with low overheads. We also compare FlashStack with Shadesmar using `Nginx` but not `Firefox`, since Shadesmar cannot build `Firefox` successfully due to register clashes. To make CPU busy and avoid I/O delays on a physical network that may mask the CPU overhead [45], `Nginx` is run on `localhost`. Its default configuration (one worker and one master) is used and the CPU affinity is set for the worker process to reduce noise introduced by switching CPU cores.

Table 2 gives the performance overheads of `Nginx` servers under FlashStack, Shadesmar, and BuddyStack in handling 10,000,000 HTTP requests from the standard `ApacheBench` tool. When

Table 2. Performance overheads of SHADESMAR, BUDDYSTACK [19] and FLASHSTACK for Nginx in handling 10,000,000 HTTP requests.

	Vanilla Nginx	SHADESMAR	BUDDYSTACK	FLASHSTACK
Concurrency	Requests/Second	Overhead		
c=1	14731.38	1.11%	2.93%	1.65%
c=2	25526.54	0.04%	0.56%	-0.30%
c=4	26077.47	-0.01%	0.32%	0.31%
c=8	26868.31	0.12%	-0.49%	-0.01%

the concurrency parameter c is 1, FLASHSTACK has a slightly higher overhead (1.65%) than SHADESMAR (1.11%) , but a smaller overhead than BUDDYSTACK (2.93%) . When $c > 1$, SHADESMAR, BUDDYSTACK and FLASHSTACK have no observable overheads. This shows that both parallel and compact shadow stacks suffer from small overheads in server programs. This holds even when the physical network delays are avoided by using `localhost`. By using the system tool `strace` to trace the system calls, we find that `Nginx` makes 11 I/O related system calls (such as `openat()` and `sendfile()`) in handling an HTTP request. Compared with these I/O systems calls, the prologue and epilogue of a protected function are relatively lightweight.

Runtime re-randomization techniques such as SHUFFLER [5] performed in code plane can protect both backward and forward edges, but at a higher overhead, as high as 15% (30%) with a shuffling period of 100 (50) milliseconds for `Nginx` [5]. In contrast, FLASHSTACK, which can protect backward edges only by shuffling in data plane, introduces only small overheads for `Nginx` (with a 10-ms shuffling policy). For the FLASHSTACK-protected `Nginx` server, the shuffling overhead introduced can be estimated as the ratio of the shuffling time over the total processing time:

$$\texttt{Shuffling Overhead} = \frac{\texttt{Total Shuffling Time}}{\texttt{Total Processing Time}} \tag{6}$$

According to Table 3, FLASHSTACK's average shuffling overhead (for its fast-moving shadow stack mechanism) is only 0.13%, costing only 12.61 μs per shuffle, on average. The average number of shuffles done per second is 99.52, approaching the maximum, 100, permitted under our 10-ms shuffling policy.

Table 3. Shuffling overheads in FLASHSTACK-protected `Nginx` when handling 10,000,000 HTTP requests.

Concurrency	Total Processing Time (secs)	Total Shuffling Time (μs)	#Shuffles	#Shuffles Per Second	Time Per Shuffle (μs)	Shuffling Overhead
c=1	690.05	846542.72	68596	99.41	12.34	0.12%
c=2	390.59	493512.45	38883	99.55	12.69	0.13%
c=4	384.66	483233.86	38294	99.55	12.62	0.13%
c=8	372.17	474540.25	37051	99.55	12.81	0.13%
Average				99.52	12.61	0.13%

In Table 4, we evaluate the shuffling overhead of our shuffling algorithm given in Figure 12 using FLASHSTACK-protected micro-benchmarks with different call stack sizes (from 256 KB to 4,096 KB). Our shuffling algorithm is repeatedly run for 1,000,000 times for each micro-benchmark. In Column 2, we set REGION_SIZE = 10 MB (which is the default used in line 15 of Figure 12 in our current implementation). In Column 3, we give the average time needed per shuffling for each physical call stack size. As can be observed, the shuffling time needed is about 11 μs for all the physical call stack sizes evaluated. As FLASHSTACK is a parallel shadow stack, the shadow stack for a call stack has exactly the same physical size as the call stack. The performance overhead of our shuffling algorithm is determined mainly by the virtual memory size (i.e., REGION_SIZE) to be remapped but generally unrelated to the underlying physical call stack size (which is usually about tens of

KBs on average for real-world programs [19]). The differences in the shuffling overheads observed between Table 3 and Table 4 are attributed to measurement errors (e.g., CPU scheduling due to the signal handling in `Nginx`) when running our shuffling algorithm.

Table 4. Shuffling overheads in FlashStack-protected micro-benchmarks with different physical call stack sizes.

Physical Call Stack Size (KB)	Virtual Memory Region Size (MB)	Time Per Shuffle (μs)
256	10	11.10
512	10	10.98
1024	10	11.04
2048	10	11.00
4096	10	11.20

Table 5. Performance overheads of BuddyStack- and FlashStack-protected `Firefox` with `Octane 2.0` and `JetStream2`.

Browser Benchmark	#Subtests	BuddyStack	FlashStack
Octane 2.0	17	5.17%	3.44%
JetStream2	64	8.84%	7.04%

For `Firefox`, Shadesmar cannot build it successfully due to register clashes. We have evaluated FlashStack and BuddyStack by running two widely-adopted browser benchmarks in Table 5. `Octane 2.0` has 17 subtests and is used for testing JavaScript engine's performance. With 64 subtests, `JetStream2` is for measuring the performance of JavaScript and WebAssembly. Even with the additional instrumentation added at call instructions for mitigating the TOCTTOU attacks, FlashStack incurs only small performance overheads, 3.44% for `Octane 2.0` and 7.04% for `JetStream2`. By comparing FlashStack and BuddyStack [19], we note that BuddyStack suffers from slightly larger overheads in protecting `Firefox`, with 5.17% for `Octane 2.0` and 8.84% for `JetStream2`.

FlashStack can build and run all the 19 C/C++ benchmarks in `SPEC CPU2006` directly. Table 6 gives the performance and code-size instrumentation overheads of these benchmarks for FlashStack and FlashStack-Sim (with FlashStack's call instruction instrumentation (Figure 11) and runtime shadow stack shuffling disabled). On average, FlashStack has a small performance overhead (5.50%) for `SPEC CPU2006`. With both its call-instruction instrumentation for mitigating the TOCTTOU attacks and its runtime shadow stack shuffling mechanism for mitigating the information disclosure attacks disabled, FlashStack-Sim has a smaller average performance overhead (4.18%). To mitigate the TOCTTOU attacks, as shown in Figure 11, FlashStack requires one fewer memory access at the prologue for a direct call but more instructions for an indirect call (than its prologue counterpart inserted at the beginning of the protected function called in Figure 10). Thus, FlashStack-Sim can sometimes have even larger performance overhead than FlashStack in some benchmarks. For FlashStack, the performance overheads of Segment+Rsp-S, dual prologues and shuffling are also accounted for. For FlashStack-Sim, only the performance overhead of Segment+Rsp-S is included. Since FlashStack exhibits only a small shuffling overhead (as revealed in Tables 3 and 4), we did not see any observable difference in the total performance overhead incurred when we only disabled shuffling in FlashStack. In other words, if we ignore the shuffling overhead in FlashStack, the performance overhead of dual prologues (introduced by instrumenting call instructions) for `SPEC CPU2006` can be approximately seen as 1.32%, which is the difference between FlashStack (5.50%) and FlashStack-Sim (4.18%).

Table 6. Performance and code-size instrumentation overheads of BuddyStack [19], FlashStack and FlashStack-Sim in SPEC CPU2006. FlashStack-Sim is the version of FlashStack with neither its call-instruction instrumentation nor its runtime shadow stack shuffling turned on

	BuddyStack		FlashStack-Sim		FlashStack	
Benchmark	Perf Overheads	File Size Increases	Perf Overheads	File Size Increases	Perf Overheads	File Size Increases
400.perlbench	10.22%	7.85%	11.56%	5.02%	11.56%	28.25%
401.bzip2	3.52%	4.01%	0.81%	3.99%	0.54%	8.01%
403.gcc	8.54%	7.00%	9.55%	4.43%	7.04%	29.87%
429.mcf	0.32%	17.95%	-1.61%	17.95%	1.93%	17.95%
433.milc	2.33%	7.12%	1.86%	4.75%	4.43%	21.36%
444.namd	1.61%	1.22%	0.40%	0.00%	0.00%	1.22%
445.gobmk	13.04%	3.75%	10.43%	2.43%	11.01%	7.10%
447.dealII	6.94%	8.19%	2.78%	5.22%	4.63%	15.03%
450.soplex	2.91%	9.78%	0.97%	6.52%	4.37%	17.11%
453.povray	19.80%	7.08%	12.87%	4.62%	15.84%	22.16%
456.hmmer	2.02%	7.70%	0.00%	5.50%	0.00%	17.61%
458.sjeng	10.34%	7.52%	7.76%	5.01%	9.77%	12.53%
462.libquantum	2.30%	13.78%	1.15%	6.89%	3.83%	20.67%
464.h264ref	8.62%	3.92%	5.17%	2.45%	4.02%	8.82%
470.lbm	2.51%	0.00%	0.50%	0.00%	3.02%	18.62%
471.omnetpp	6.47%	12.97%	2.27%	8.48%	5.83%	32.41%
473.astar	5.76%	6.72%	3.03%	6.71%	3.03%	13.53%
482.sphinx3	5.68%	8.74%	2.70%	5.25%	2.16%	24.48%
483.xalancbmk	12.65%	11.50%	7.23%	7.22%	11.45%	28.75%
Average	6.61%	7.73%	4.18%	5.39%	5.50%	18.18%

Table 7. Statistics collected for `Nginx1.18` and `Firefox79.0`.

Benchmark		M (GB)	h	#Faults	Time (secs)	k_{max}
Nginx		0.04	3.05×10^{-7}	0	372.17	1
Firefox	Octane 2.0	4.27	3.26×10^{-5}	1	33.18	2
	JetStream2	23.95	1.83×10^{-4}	12	316.19	2

Due to compatibility problems, Shadesmar cannot run `447.dealII`, `453.povray`, `471.omnetpp` and `483.xalancbmk` successfully. As for the other 15 C/C++ benchmarks, Shadesmar has an average performance overhead of 4.88%, which is slightly smaller than in BuddyStack [19] (5.32%) but still slightly larger than in FlashStack (4.45%) and FlashStack-Sim (3.62%). Note that the 18 C/C++ SPEC CPU2006 benchmarks (excluding `447.dealII`) are used to evaluate Shadesmar [10] on Ubuntu 16.04. The version of their system libraries may be different from ours (`ld-2.27.so` and `libc-2.27.so`). This may explain why they did not encounter any program crash in their evaluation, but this also suggests that reserving a general register to maintain the shadow stack pointer can lead to platform-dependent compatibility problems that are even harder to resolve.

Let us now examine the code-size instrumentation overheads. On average, even with the call instructions instrumented to mitigate the TOCTTOU attacks, the average code size increase under FlashStack (18.18%) is still a little smaller than that under Shadesmar (20.38%). In the case of FlashStack-Sim, however, this has dropped to 5.39%. As for BuddyStack [19], we have 7.73%. Note that both Shadesmar and BuddyStack are vulnerable to the TOCTTOU attacks.

6.2 Security

We explain and analyze how FlashStack can be used to mitigate existing attacks (such as information disclosure and crash-resistant attacks [12, 13, 25, 30]) for exposing the locations of its shadow stacks. Let us start by considering the well-known thread spraying and memory oracles attacks [12, 13]. To expose hidden safe regions, they reduce the entropy of the 47-bit user space by consuming a large amount of virtual memory space. However, as reported in [29], programs

rarely consume terabytes of memory and even big data applications only use gigabytes of virtual memory space. This is consistent with the findings in our evaluation. As shown in Table 7, `Nginx` consumes only 0.04 GB virtual memory space in handling the HTTP requests from `ApacheBench`, while `Firefox` needs only 4.27 GB for `Octane 2.0` and only 23.95 GB for `JetStream2`.

FlashStack mitigates these attacks in `get_memory_ at_random()` of Figure 13 (Section 5.3). In lines 28-29, if `k > THRESHOLD`, where `k` records the maximum number of attempts made in allocating the memory required for a new shadow stack (by calling `mmap()`), FlashStack will call `CHECK_MAPPED_REGIONS()` to check the total size of mapped memory regions available. If this happens to be larger than a reasonable size predefined, FlashStack will raise the alarm for a possible entropy-reduction attack and stop the program. Currently, `THRESHOLD = 6` is set empirically.

We discuss below why this particular value of `THRESHOLD` is selected analytically. Let N be the total number of shuffles requested made, i.e., the number of times that `get_memory_at_random()` is called. Let `M` be the total size of mapped memory regions in a process (in GB). Note that `M` is usually no more than 1024 GB (1 TB) for a protected process. Let `S` $= 128 \times 1024$ GB (128 TB) be the total size of the 47-bit user space. Let h be the ratio of `M` over `S`:

$$h = \frac{M}{S} \tag{7}$$

In line 9 of `get_memory_at_random()`, the probability P_k for generating a random value `x` that does not hit a mapped region in the k-th attempt for the first time can be expressed as:

$$P_k = (1 - h) \times h^{k-1} \tag{8}$$

To perform a worst-case analysis, we assume effectively that `MAP_FIXED` is used in the `mmap()` call in Figure 13. Note that shuffling the shadow stacks around in a process does not change the number of mapped regions (assuming the number of threads in the process is roughly fixed).

Let A_k be the number of calls to `get_memory_at_random()` that can allocate a new shadow stack at the k-th attempt:

$$A_k \approx N \times P_k \tag{9}$$

Thus, the ratio of A_{k+1} over A_k can be expressed by:

$$\frac{A_{k+1}}{A_k} \approx h \tag{10}$$

Finally, we write $k_{\max}$ for the largest k such that $A_k \geqslant 1$.

In `get_memory_at_random()` (Figure 13), we can collect these statistics easily. The number of attempts, i.e., calls to `mmap()` in lines 16-18 is recorded in `k` (declared in line 5) and A_k can be recorded using $A[k]$ (declared in line 1).

Table 8. The distributions of A_k (and $k_{\max}$) in a test program as M (GB) varies, where $N = 2,560,000$ and $S = 128 \times 1024$ GB.

A_k	$h = \frac{1}{4096}$ $M = 32$	$h = \frac{1}{2048}$ $M = 64$	$h = \frac{1}{1024}$ $M = 128$	$h = \frac{1}{512}$ $M = 256$	$h = \frac{1}{256}$ $M = 512$	$h = \frac{1}{128}$ $M = 1024$
A_1	2,559,340	2,558,641	2,557,387	2,554,851	2,549,802	2,539,811
A_2	660	1,358	2,611	5,141	10,147	20,031
A_3		1	2	8	51	157
A_4						1
$k_{\max}$	2	3	3	3	3	4

Table 8 gives the distributions of A_k ($k_{\max}$) in a test program as M varies from 32 GB to 1024 GB, where $N = 2,560,000$. Even when `M` reaches 1024 GB, in which case, $h = \frac{1}{128}$, FlashStack still

succeeds in allocating a new shadow stack with only one attempt for the majority of the shuffles requested (since $P_1 = 1 - h = 99.21\%$) and no more than four attempts in the worst case (since $k_{\max} = 4$).

Table 9. The distributions of A_k (and $k_{\max}$) as N varies, where $M \in \{32\text{ GB}, 1024\text{ GB}\}$ and $S = 128 \times 1024$ GB.

	N= 2,560,000		N = 25,600,000		N = 256,000,000		N = 2,560,000,000	
A_k	M = 32, $h = \frac{1}{4096}$	M = 1024, $h = \frac{1}{128}$	M = 32, $h = \frac{1}{4096}$	M = 1024, $h = \frac{1}{128}$	M = 32, $h = \frac{1}{4096}$	M = 1024, $h = \frac{1}{128}$	M = 32, $h = \frac{1}{4096}$	M = 1024, $h = \frac{1}{128}$
A_1	2,559,340	2,539,811	25,593,166	25,396,588	255,931,616	253,969,797	2,559,315,894	2,539,704,307
A_2	660	20,031	6,833	201,778	68,373	2,014,222	683,927	20,134,418
A_3		157	1	1,621	11	15,868	179	160,084
A_4		1		13		112		1,185
A_5						1		6
$k_{\max}$	2	4	3	4	3	5	3	5

In Table 9, we examine the distributions of A_k ($k_{\max}$) as N varies from 2,560,000 to 2,560,000,000, where $M \in \{32\text{ GB}, 1024\text{ GB}\}$. Consider the case when $N = 2,560,000,000$. If $M = 32$ GB, then $k_{\max} \leqslant 3$. If M increases to 1024 GB, then $k_{\max}$ increases to 5 only. Even if we increase M further to 2048 GB (not shown), then $k_{\max} \leqslant 6$.

Based on our analysis (Equations 7 – 10) and our validation (Tables 8 and 9), we have therefore set `THRESHOLD` = 6 in our current implementation. As discussed above, FlashStack can apply such a simple inspection mechanism provided in lines 28-29 of `get_memory_at_random()` to mitigate entropy-reduction-related attacks [12, 13] when it shuffles its shadow stacks continuously.

Next, let us discuss how FlashStack mitigates crash-resistant attacks targeting `Firefox`. To enhance performance, its `asm.js` optimization module uses exceptions rather than explicit bounds checking [30], as any access in bounds does not lead to an exception. However, this optimization is exploited in CROP [30] to obtain crash-resistance in browsers like `Firefox`. CROP can launch 18,357 crash-resistant probes (faults) per second under 64-bit Linux.

However, as shown in Table 7, during its normal execution, `Firefox` exhibits only one segment fault for `Octane 2.0` in 33.18 seconds and 12 segment faults for `JetStream2` in 316.19 seconds. As out-of-bounds accesses are usually rare, 18,357 segment faults per second are obviously an abnormal behavior. In FlashStack, we instrument the signal handler `WasmTrapHandler()` in `Firefox79.0` to monitor the number of faults raised per second. If it is higher than a preset threshold (20 in our current setting), we treat it as an alarm that the browser is being probed by CROP attacks. Currently, we set REGION_SIZE and STK_SIZE in Figure 7 to be 10 and 8 MB, respectively. The entropy of our shadow stack is $\frac{128 \times 2^{20}}{10}$ (i.e., about 23 bits). CROP [30] can achieve 18,357 probes per second in 64-bit Linux, requiring about $\frac{2^{23}}{18,357} = 457$ seconds to locate the shadow stack in browsers. By slowing down the malicious crash-resistant probing to be no more than 20 faults per second, 419,430 seconds (6,990.5 minutes) will be needed, making it much harder for the attacker to launch CROP attacks before an alarm is raised (due to ongoing probes). In addition, in FlashStack, the shadow stack is shuffled under a 10-ms shuffling policy. Thus, we can make the knowledge accumulated by CROP probes obsolete, thus making it much harder for the shadow stack to be located by the attacker. Note that no CROP attacks exist in server programs [30].

Our evaluation also shows that the program in Figure 2 can be protected by FlashStack to thwart the TOCTTOU attack proposed by Microsoft's red team [1], as `StrnLen()` (as a protected function) will never use the return address on the call stack any more.

As AnC [25] needs to allocate an abnormally large virtual memory region (4 TB), it can be blocked when such an abnormal behavior is detected. Our 10-ms shuffling policy can remap the shadow stack to make it move continuously, thus mitigating the side-channel attacks based on page-table walks.

Although we use a 10-ms shuffling policy in protecting `Nginx` and `Firefox`, increasing 10 ms to say, 20-100 ms can still be reasonable [5]. It means that there is still enough leeway for FlashStack to strike a good balance between security and performance even when FlashStack is applied to systems with a large number of CPU cores. Admittedly, the attacker may find some faster attacks against browsers. Thus, the protection provided by FlashStack for browsers is meaningful, but weaker than that for servers.

6.3 Compatibility

FlashStack can build and run `Nginx` and `SPEC CPU2006` successfully. To run `Firefox` under FlashStack, however, some configurations are needed. First, as FlashStack relies on `arch_prctl()` and `mremap()` to shuffle shadow stacks, we need to allow these system calls to be made under the sandbox policy of `Firefox` [46] during the shuffling period. Second, we also need to disable the `SA_ONSTACK` flag, as `sigaltstack()` has not been supported yet [47]. Third, for some customized systems where `ARCH_SET_GS` [23, 24] is disabled, their administrators need to enable `ARCH_SET_GS`. For mainstream 64-bit Linux platforms such as Ubuntu, `ARCH_SET_GS` is enabled by default.

Given its complexity (with 21 million lines of code [48]), `Firefox` can act as an excellent benchmark for measuring the degrees of compatibility and deployability provided by a hardening tool. By running `Firefox` successfully under FlashStack, we believe that FlashStack can provide reasonable compatibility for most of the user-space C/C++ programs, but no full compatibility [49].

6.4 Discussions

Currently, as in Shadesmar [10], we do not consider just-in-time code generation [7]. Our current shuffling approach is implemented in the user space, implying that that it may be maliciously paused by the attacker. To raise the bar higher, a more secure alternative can be implemented in the kernel space. In addition, making the shadow stack of a blocked/sleeping thread read-only (by the thread scheduler) can further mitigate cross-thread attacks even when a thread is maliciously blocked. Finally, our shuffling policy does not guarantee that a hooked library function is called exactly every 10 ms. In fact, our 10-ms shuffling policy only guarantees that there are no more than 100 shufflings per second in a thread. Having a dedicated shuffling thread may be potentially more secure but will require frequent inter-thread synchronizations.

The basic idea behind CFI [6–8] is simple but hard to enforce in practice [1]. To the best of our knowledge, source- and binary-level CFI techniques are often discussed separately in the literature [6, 50–55]. As long as some unprotected (non-instrumented) functions are executed (which can happen when, for example, either an indirect call in a protected function is redirected maliciously to an unprotected function or a protected function may be called in unprotected libraries via callbacks), all existing (coarse-grained or fine-grained) source-level CFI mechanisms may need to be enhanced by some binary-level CFI techniques in order to provide the CFI protection as promised [54, 55]. Binary instrumentation [56, 57] and disassembly [52–55] can be used to provide better protection for non-instrumented libraries. Incremental deployment [26, 27] is essential for a hardening tool to be adopted, as it represents a practical trade-off among security, performance and compatibility. If performance can be disregarded, dynamic binary instrumentation techniques [56, 57] can be used to reinterpret existing instructions and add additional semantics needed for security, but such solutions often suffer from high performance overhead. Similarly, if compatibility can be ignored, a theoretically promising design will unlikely be deployable on existing computing platforms (due to the incompatibility problem discussed earlier).

A robust and complete control-flow integrity framework needs to combine software-based (or hardware-enforced) shadow stacks, source-level CFI techniques, and binary instrumentation (or disassembly) together. This challenging research problem will have to be addressed in future work.

7 RELATED WORK

We review the past relevant work, by discussing call stack defenses, CFI, and randomization.

Call Stack Defenses. To tackle stack smashing attacks [58], many defenses [8–10, 18, 59–63] have been proposed to protect return addresses on the call stack. Earlier defenses [18, 59, 60] focus on counteracting traditional stack buffer overflow attacks, but are vulnerable to arbitrary writes. Later defenses like SafeStack [61] aim to hide the call stack in a huge virtual memory space, but are still vulnerable to information disclosure attacks [12, 13]. SafeHidden [29] provides an effective mechanism for shuffling safe regions, but it needs virtualization or a thin hypervisor. In addition, the shadow stack used in SafeHidden is inefficient (similar as LLVM's ShadowCallStack) and vulnerable to the TOCTTOU attacks. As its shuffling is driven by `mmap()`, the SafeHidden-protected `Nginx` may not trigger randomization timely in handling I/O requests which only access allocated memory space [29]. The combination of FlashStack and SafeHidden may be a possible option in cloud computing. Shadesmar is a state-of-the-art compact shadow stack design, but it uses a dedicated register `r15` to maintain the shadow stack pointer, compromising compatibility with unprotected code. A leak-resilient dual stack is proposed in [63] for protecting return addresses with negligible overheads, but its reliance on a general register also sacrifices its compatibility. Microsoft's RFG has been deprecated due to the TOCTTOU attacks [1, 22].

Compared with FlashStack, BuddyStack [19] can also provide protection for multi-threaded programs but BarRA [8] supports only single-threaded programs. To avoid register clashes, BuddyStack uses a stack-based thread-local storage mechanism and exploits also network-access delays to re-randomize return addresses (without shuffling the shadow stack) to protect server programs. As a result, its protection for browsers is limited. In contrast, FlashStack reserves a segment register to enforce the integrity of the shadow stack by shuffling it continuously in user space, thereby providing strong protection for servers (as in BuddyStack) but also better security for browsers, but at the expense of sacrificing some compatibility on x86-64. In addition, FlashStack relies on MMU to shuffle the shadow stack, but this may not be available on embedded systems (e.g., 32-bit Cortex M3/M4 processors [64]).

Control-Flow Integrity. CFI [6, 50, 51] is an effective mechanism to protect forward edges. There are two types of CFI schemes: coarse-grained [65–67] and fine-grained [32, 33, 68]. Shadow stacks are often used for protecting backward edges. However, as reported in [1, 17], shadow stacks that can provide performance, security and compatibility on x86-64 are still hard to obtain. We hope FlashStack can serve as a good starting point and will open-source it to facilitate further studies.

Randomization. There are also two classes of randomization mechanisms: coarse-grained [3] and fine-grained [69–74]. However, even fine-grained randomization techniques are still vulnerable to JIT-ROP attacks [75]. To mitigate JIT-ROP or clone-probing attacks [4, 75, 76], runtime code re-randomization [4, 5, 77] has been proposed. But pointer tracking is time-consuming for code re-randomization [4]. Code sharing is another challenging issue in software diversity [78]. In contrast, FlashStack only shuffles shadow stacks in data plane and does not need to track code pointers. Thus, FlashStack-protected code sections can still be shared as usual among different processes.

8 CONCLUSION

We introduce FlashStack as a software-based shadow stack design for x86-64. By designing a novel mapping mechanism, detecting entropy-reduction attacks during shuffling, and thwarting the TOCTTOU attacks with a dual-prologue instrumentation scheme, we show that FlashStack

represents a simple and practical shadow stack design for x86-64 in terms of performance, security and compatibility provided.

REFERENCES

[1] Microsoft Security Response Center, "The evolution of CFI attacks and defenses," https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2018_02_OffensiveCon, 2018.

[2] Wikipedia, "Tiger lake," https://en.wikipedia.org/wiki/Tiger_Lake_(microprocessor), 2020.

[3] The PaX Team, "Address space layout randomization," https://pax.grsecurity.net/docs/aslr.txt, 2001.

[4] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make ASLR win the clone wars: Runtime re-randomization," in *Network and Distributed System Security Symposium*, 2016, pp. 1–15.

[5] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 367–382.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM SIGSAC Conference on Computer and Communications Security*, 2005, pp. 340–353.

[7] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1317–1328.

[8] C. Zou and J. Xue, "Burn after reading: A shadow stack with microsecond-level runtime rerandomization for protecting return addresses," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020, pp. 258–270.

[9] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 555–566.

[10] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019, pp. 985–999.

[11] K. Chen and D. Wagner, "Large-scale analysis of format string vulnerabilities in debian linux," in *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, 2007, pp. 75–84.

[12] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 121–138.

[13] E. Goktas, A. Oikonomopoulos, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Bypassing Clang's SafeStack for Fun and Profit," in *Black Hat Europe*, 2016, pp. 1–76.

[14] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 161–176.

[15] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 401–416.

[16] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*. IEEE, 2016, pp. 179–194.

[17] T. C. Team, "ShadowCallStack," https://clang.llvm.org/docs/ShadowCallStack.html, 2021.

[18] T.-c. Chiueh and F.-H. Hsu, "Rad: A compile-time solution to buffer overflow attacks," in *Proceedings 21st International Conference on Distributed Computing Systems*, 2001, pp. 409–417.

[19] C. Zou, X. Wang, Y. Gao, and J. Xue, "Buddy stacks: Protecting return addresses with efficient thread-local storage and runtime re-randomization," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, pp. 1–37, 2022. [Online]. Available: https://doi.org/10.1145/3494516

[20] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy*, 2016, pp. 969–986.

[21] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1868–1882.

[22] M. Branscombe, "Windows 10 security: How the shadow stack will help to keep the hackers at bay," https://www.techrepublic.com/article/windows-10-security-how-the-shadow-stack-will-help-to-keep-the-hackers-at-bay/, 2020.

[23] L. K. Documents, "Using fs and gs segments in user space applications," https://www.kernel.org/doc/html/latest/x86/x86_64/fsgs.html, 2021.

[24] L. M. Pages, "Set architecture-specific thread state," https://man7.org/linux/man-pages/man2/arch_prctl.2.html, 2021.

[25] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *Network and Distributed System Security Symposium*, 2017, pp. 1–15.

[26] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, "CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1805–1821.

[27] A. Prakash and H. Yin, "Defeating rop through denial of stack pivot," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 111–120.
[28] L. manual page, "getcontext(3)," https://man7.org/linux/man-pages/man3/getcontext.3.html.
[29] Z. Wang, C. Wu, Y. Zhang, B. Tang, P.-C. Yew, M. Xie, Y. Lai, Y. Kang, Y. Cheng, and Z. Shi, "Safehidden: an efficient and secure information hiding technique using re-randomization," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1239–1256.
[30] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling client-side crash-resistance to overcome diversification and information hiding." in *Network and Distributed System Security Symposium*, 2016, pp. 1–15.
[31] H. Tankovska, "Daily time spent on social networking by internet users worldwide from 2012 to 2020," https://www.statista.com/statistics/433871/daily-social-media-usage-worldwide/, 2021.
[32] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "VTrust: Regaining trust on virtual calls," in *Network and Distributed System Security Symposium*, 2016, pp. 1–15.
[33] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 914–926.
[34] Microsoft, "Data execution prevention," https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention, 2018.
[35] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 973–990.
[36] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, 2019, pp. 1–19.
[37] J. Sun, X. Zhou, W. Shen, Y. Zhou, and K. Ren, "PESC: A per system-call stack canary design for linux kernel," in *Proceedings of the 10th ACM Conference on Data and Application Security and Privacy*, 2020, pp. 365–375.
[38] D. Ulrich, "Elf handling for thread-local storage," https://akkadia.org/drepper/tls.pdf, 2013.
[39] M. Zalewski, "American fuzzy lop (AFL) fuzzer," http://lcamtuf.coredump.cx/afl/, 2021.
[40] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.
[41] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.
[42] E. Itkin, "Bypassing return flow guard," https://eyalitkin.wordpress.com/2017/08/18/bypassing-return-flow-guard-rfg/, 2017.
[43] mmap, "Linux manual page," https://man7.org/linux/man-pages/man2/mmap.2.html.
[44] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys," in *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019, pp. 241–254.
[45] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida, "SoK: Benchmarking flaws in systems security," in *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, 2019, pp. 310–325.
[46] M. Wiki, "Security/sandbox/seccomp," https://wiki.mozilla.org/Security/Sandbox/Seccomp, 2016.
[47] L. Documents, "Safe stack," https://clang.llvm.org/docs/SafeStack.html, 2021.
[48] B. Abadie and S. Ledru, "Engineering code quality in the firefox browser," https://hacks.mozilla.org/2020/04/code-quality-tools-at-mozilla/, 2020.
[49] WINE, "What is wine?" https://www.winehq.org/, 2021.
[50] M. Muench, F. Pagani, Y. Shoshitaishvili, C. Kruegel, G. Vigna, and D. Balzarotti, "Taming Transactions: Towards Hardware-Assisted Control Flow Integrity Using Transactional Memory," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defense (RAID)*, 2016, pp. 24–48.
[51] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding cracks in shields: On the security of control flow integrity mechanisms," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1821–1835.
[52] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 1187–1198.
[53] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Network and Distributed System Security Symposium*, 2018, pp. 1–15.
[54] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 1075–1092.
[55] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level," in *Network and Distributed System Security Symposium, NDSS*, 2021.

[56] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.
[57] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 89–100.
[58] A. One, "Smashing the stack for fun and profit," http://phrack.org/issues/49/14.html, 1996.
[59] S. Shield, "A stack smashing technique protection tool for linux," https://www.angelfire.com/sk/stackshield/info.html, 2000.
[60] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium*, 1998, pp. 1–16.
[61] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014, pp. 147–163.
[62] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries." in *NDSS*, 2015, pp. 1–15.
[63] P. Zieris and J. Horsch, "A leak-resilient dual stack scheme for backward-edge control-flow integrity," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 369–380.
[64] STMicroelectronics, "Stm32 32-bit arm cortex mcus," https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html.
[65] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013, pp. 559–573.
[66] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 447–462.
[67] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against rop attack," in *Network and Distributed System Security Symposium*, 2014, pp. 1–14.
[68] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 941–955.
[69] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003, pp. 272–280.
[70] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006, pp. 339–348.
[71] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.
[72] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 280–291.
[73] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 40–55.
[74] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.
[75] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013, pp. 574–588.
[76] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014, pp. 227–242.
[77] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 268–279.
[78] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, 2018, pp. 461–477.