Buddy Stacks: Protecting Return Addresses with Efficient Thread-Local Storage and Runtime Re-Randomization

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

Shadow stacks play an important role in protecting return addresses to mitigate ROP attacks. Parallel shadow stacks, which shadow the call stack of each thread at the same constant offset for all threads, are known not to support multi-threading well. On the other hand, compact shadow stacks must maintain a separate shadow stack pointer in **thread-local storage (TLS)**, which can be implemented in terms of a register or the per-thread **Thread-Control-Block (TCB)**, suffering from poor compatibility in the former or high performance overhead in the latter. In addition, shadow stacks are vulnerable to information disclosure attacks.

In this paper, we propose to mitigate ROP attacks for single- and multi-threaded server programs running on general-purpose computing systems by using a novel stack layout, called a ***buddy stack*** (referred to as **BUSTK**), that is highly performant, compatible with existing code, and provides meaningful security. These goals are met due to three novel design aspects in BUSTK. First, BUSTK places a parallel shadow stack just below a thread's call stack (as each other's buddies allocated together), avoiding the need to maintain a separate shadow stack pointer and making it now well-suited for multi-threading. Second, BUSTK uses an efficient stack-based thread-local storage mechanism, denoted STK-TLS, to store thread-specific metadata in two TLS sections just below the shadow stack in dual redundancy (as each other's buddies), so that both can be accessed and updated in a lightweight manner from the call stack pointer `rsp` alone. Finally, BUSTK re-randomizes continuously (on the order of milliseconds) the return addresses on the shadow stack by using a new microsecond-level runtime re-randomization technique, denoted STK-MSR. This mechanism aims to obsolete leaked information, making it extremely unlikely for the attacker to hijack return addresses, particularly against a server program that sits often tens of milliseconds away from the attacker.

Our evaluation using web servers, `Nginx` and `Apache Httpd`, shows that BUSTK works well in terms of performance, compatibility, and security provided, with its parallel shadow stacks incurring acceptable memory overhead for real-world applications and its STK-TLS mechanism costing only two pages per thread. In particular, BUSTK can protect the `Nginx` and `Apache` servers with an adaptive 1-ms re-randomization policy (without observable overheads when IO is intensive, with about 17,000 requests per second). In addition, we have also evaluated BUSTK using other non-server applications, `Firefox`, `Python`, `LLVM`, `JDK` and `SPEC CPU2006`, to demonstrate further the same degree of performance and compatibility provided, but the protection provided for, say, browsers, is weaker (since network-access delays can no longer be assumed).

CCS Concepts: • **Security and privacy → Software and application security;**

Additional Key Words and Phrases: Shadow stack, buddy stack, runtime re-randomization, CFI, ROP

This work has been supported by Australian Research Council Grants (DP170103956 and DP180104069).
Authors' addresses: C. Zou, X. Wang, and J. Xue, UNSW Sydney, Kensington, Sydney, New South Wales, 2052, Australia; emails: {changweiz, xudongw}@cse.unsw.edu.au, j.xue@unsw.edu.au; Y. Gao, Huawei Toronto Research Center, Suite 300, 19 Allstate Parkway, Markham, Ontario, L3R 5A4, Canada; email: yaoqing.gao@huawei.com.

1049-331X/2022/02-ART35e $15.00
https://doi.org/10.1145/3494516

ACM Reference format:
Changwei Zou, Xudong Wang, Yaoqing Gao, and Jingling Xue. 2022. Buddy Stacks: Protecting Return Addresses with Efficient Thread-Local Storage and Runtime Re-Randomization. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 35e (February 2022), 37 pages.
https://doi.org/10.1145/3494516

1 INTRODUCTION

Code reuse attacks [7, 14, 65] such as **return-oriented programming (ROP)** are still a major security threat to computer systems, especially since most of the fundamental software projects (e.g., web servers, browsers, and JVM) are coded in C and C++, which lack both memory safety and type safety [38, 55, 56, 74, 95]. With the rapid development of **control flow integrity (CFI)** [1, 11, 33, 40, 58, 88, 89] in protecting forward edges, i.e., function pointers and virtual table pointers, backward edges, i.e., return addresses are increasingly becoming a favorite target for control-flow hijacking [12]. Although Intel **Control-Flow Enforcement Technology (CET)** has been released recently to provide the hardware support for shadow stacks [80], its market penetration is estimated to take more than six years [13]. Hundreds of millions of existing desktops, servers and embedded systems still need to be protected without the hardware support of Intel CET. Therefore, shadow stacks [12, 21, 84], which are often assumed to be used to protect return addresses in the previous research on CFI [1, 11, 33, 40, 58, 88, 89], need to be carefully revisited.

There are two types of shadow stacks, parallel shadow stacks [12, 17, 21, 33, 84], which place the shadow stack at a constant offset of the original call stack, and compact shadow stacks [12, 17], which maintain a separate shadow stack pointer to a smaller shadow stack that is just large enough to store the return addresses on the call stack. We review below representative shadow stack designs in these two categories along three axes, performance, compatibility, and security, in protecting real-world multi-threaded programs (including web servers as an important special case), by highlighting their advances made and limitations in meeting these goals.

Prior Work. Parallel shadow stacks [12, 17, 21, 33, 84] effectively use the call stack pointer as the shadow stack pointer (by adding a constant offset). This design is thus highly efficient, compatible with existing code, and supports trivially stack unwinding. However, requiring all the call stacks to be shadowed at the same constant offset will severely constrain the address space layout for a program with many threads [12], making this design ill-suited for multi-threading. Due to nondeterminism in thread-scheduling, a thread may find that the space needed by its shadow stack (at the given fixed offset) has already been taken by other threads. In addition, parallel shadow stacks are perceived to incur high memory overhead in the literature as it requires each call stack to be duplicated (but this perception is not necessarily true as shown in this paper).

Compact shadow stacks [12, 17] store return addresses in a smaller shadow stack by maintaining a separate shadow stack pointer. Thus, this design is memory-efficient and well-suited for multi-threading, but incurs additional instrumentation in handling stack unwinding [12]. In addition, depending on the types of **thread-local storage (TLS)** used for storing the shadow stack pointer, two schemes exist: (1) REG-TLS, in which a dedicated REGister is reserved and used, and (2) TCB-TLS, in which the per-thread **Thread-Control-Block (TCB)** is used. For REG-TLS, its reserved register is often accessed directly via inline assembly instructions. For TCB-TLS, a keyword, `__thread`, has been added to C and C++ to allow TCB-TLS variables to be declared [77].

Unlike parallel shadow stacks, compact shadow stacks must pay for the additional cost incurred in maintaining the shadow stack pointer in these two schemes. In addition, REG-TLS sacrifices compatibility with existing code (due to register clashes) while TCB-TLS suffers from high

```
01 unprotected_foo:
02   push   %r15  # Information leakage
03   mov    $0,%r15
04   # call protected_bar
05   call   *%rdi
06   pop    %r15
07   retq

08 protected_bar:
09   # Shadesmar prologue
10   mov    (%rsp),%rax
11   mov    %rax,(%r15) # Program crash
12   mov    %rsp,0x8(%r15)
13   lea    0x10(%r15),%r15
14   ...
```

(a) REG-TLS

```
# API
15 void *__tls_get_addr(tls_index *ti);
# Define a TCB-TLS variable, which can
# be the per-thread shadow stack pointer.
16 __thread long var;

# Access TCB-TLS from PIC
17 tls_in_pic:
18   .byte 0x66
19   lea  var@tlsgd(%rip), %rdi
20   .value  0x6666
21   rex64
22   call  __tls_get_addr@PLT
23   mov  (%rax), %rax
24   ...
```

(b) TCB-TLS

Fig. 1. Illustrating and comparing REG-TLS and TCB-TLS. (a) REG-TLS (where r15 is used to store the shadow stack pointer as in Shadesmar [12]): information leakage (line 2) and a program crash (line 11) and (b) TCB-TLS: high performance overhead when accessing a TCB-TLS variable in PIC.

performance overhead (due to frequent and expensive accesses to TCB). Below let us examine these two types of shadow stack designs, as illustrated in Figure 1.

- **REG-TLS.** Shadesmar [12] represents the state-of-the-art solution, which uses a dedicated register (i.e., r15 on ×86-64) to store the shadow stack pointer. Due to incompatibility with shared libraries, Shadesmar can build Apache Httpd but fails to run it due to the program crashes caused by register clashes. In addition, Shadesmar cannot build Firefox, Clang/LLVM, the Python interpreter, and JDK successfully from source code (as validated in Section 5.2), leading to program crashes in automated testing (during the build). According to its authors, compatibility is essential to ensure whether a shadow stack design can be incrementally accepted and deployed or not, in practice.

  Let us examine this compatibility issue, as illustrated in Figure 1(a), where register r15 is reserved to save the shadow stack pointer in protected code. Suppose the function protected_bar() is hardened by Shadesmar [12]. The prologue in lines 10-13 is generated to store its return address and the call stack pointer rsp on the shadow stack pointed by r15. If protected_bar() is passed as the first argument to the unprotected function, unprotected_foo(), according to the System V ABI [50], register rdi will be used to store this argument. To comply with the calling convention, unprotected_foo() needs to save r15 on the call stack in line 2 before storing an immediate value 0 in this register in line 3. The call instruction in line 5 will invoke protected_bar(), which assumes that r15 is a dedicated register for storing the shadow stack pointer, but its value has been modified by unprotected_foo() in line 3, leading to a program crash (caused by a null-pointer dereference) in line 11.

  In the presence of unprotected code, REG-TLS is also vulnerable to information disclosure attacks. If r15 points to the shadow stack in line 2, then its location, which is saved on the call stack, can be leaked (before it is overwritten). This example demonstrates that REG-TLS (with Shadesmar [12] as a state-of-the-art instantiation) both sacrifices compatibility (with existing code) and fails to provide the integrity guarantees for the shadow stack.

Worse still, these two issues cannot be easily eliminated even if the source code for a program is available. For example, our evaluation reveals that although `Firefox79.0` is mostly implemented in C, C++, and Rust, there are still 584 assembly source files. The presence of the hard-coded register `r15` in some assembly files will still cause register clashes for SHADESMAR, unless all these hand-crafted assembly files are totally rewritten. In practice, open-source browsers like `Firefox` also use some closed-source plugins [2, 52]. This observation also carries over to server programs.

Making SHADESMAR (a state-of-the-art instantiation of REG-TLS) compatible with unprotected code cannot be easily achieved by modifying the linker to add wrapper functions to save `r15` for calls across the protection boundary, as suggested in [12]. The reasons for this are two-fold. First, the state-of-the-art pointer analyses for C/C++ such as SVF [72, 73] are applicable only to standard-compliant C/C++ source code and scalable for only moderate-sized multi-threaded codebases [69, 71, 91]. As a result, determining which protected functions are callbacks can only be done either imprecisely (hurting the performance of SHADESMAR) or unsoundly (failing to improve compatibility for SHADESMAR). Second, how to save and restore `r15` without information leakage is another challenging issue.

- **TCB-TLS.** Unlike REG-TLS, TCB-TLS makes the opposite tradeoff by trading high performance overhead for compatibility. Instead of storing the shadow stack pointer in a register as in REG-TLS, TCB-TLS will store it as a TCB-TLS variable in the thread-specific TCB. Unfortunately, TCB-TLS will be too expensive to be deployable in practice [18].

  Figure 1(b) shows how `var`, a TCB-TLS variable declared in line 16, is accessed in **position independent code (PIC)**, which is often used in shared libraries, so that it can be mapped to different locations in different processes. The instructions in lines 18-23 are needed to access `var` in PIC on ×86-64 [77]. The library function `__tls_get_addr()` is called to return the address of a thread-local variable (identified by its `tls_index`) with its signature given in line 15. The required argument is first moved to `rdi` (line 19) and `__tls_get _addr()` is then called (line 22), followed by a dereference to load the value of `var` (line 23). Lines 18, 20, and 21 are prefixes to increase the code sequence to 16 bytes [77]. If `var` represents the per-thread shadow stack pointer, then the code sequence that is similar to that in lines 18-23 will be frequently executed, resulting in a high performance overhead, as `__tls_get_addr()` will be called twice in a protected function (at both its prologue and epilogue). Our evaluation shows that TCB-TLS is one order of magnitude slower than REG-TLS when accessed from PIC (Section 3.1.3). TCB-TLS relies on information hiding in a large virtual space, but information hiding is still vulnerable to information disclosure attacks. Therefore, TCB-TLS itself is not good enough for protecting return addresses. Due to its deficiency in both performance and security, to the best of our knowledge, no open-source TCB-TLS-based shadow stacks exist.

  Finally, the integrity of shadow stacks must be enforced in order to provide meaningful security. Existing mitigation mechanisms include (1) ASLR [75], (2) runtime code re-randomization [5, 47, 82], (3) runtime return address re-randomization [96], and (4) restricted access privileges [12]. ASLR, which relies on information hiding, performs a single, per-process randomization that is applied before or at the process' load-time, but is still vulnerable to information disclosure attacks [32, 59]. Runtime code re-randomization [5, 47, 82] mitigates the threat of information disclosure by performing continuous code re-randomization at runtime, but pointer tracking is time consuming and troublesome [47]. BARRA [96] avoids pointer tracking by re-randomizing the return IDs abstracted from return addresses. This scheme thus requires whole program analysis in order to achieve compatibility with closed-source libraries, but at the expense of failing to protect the return addresses of functions

potentially invokable from closed-source libraries. In practice, BarRA may fail to protect a large number of functions as state-of-the-art pointer analyses for multi-threaded C/C++ programs are either unscalable or imprecise, especially since they are required to compute the points-to facts soundly [71]. In addition, BarRA does not support multi-threading well (as it uses a parallel shadow stack at the same constant offset from the call stack) and thus cannot be used to protect server programs like `Apache Httpd`. Finally, Shadesmar [12] encodes access privileges by using Intel ×86 extensions for memory protection (MPX [35]) and page table control (MPK [63]) to guarantee the integrity of shadow stacks, but neither exhibits performance "numbers acceptable for a deployed mechanism" according to its authors.

This Work. In this paper, we propose to protect return addresses for single- and multi-threaded server programs running on general-purpose computing systems to mitigate ROP attacks by using a novel stack layout, called a *buddy stack* (referred to as Bustk), that is highly performant, compatible with existing code, and provides meaningful security. Bustk is also designed to work with ASLR [75] seamlessly. For each thread, we reserve a single virtual space that consists of its call stack and its parallel shadow stack (as each other's buddies), as well as its two metadata sections (also as each other's buddies) as its **TLS (Thread-Local Storage)** in dual redundancy.

To satisfy these design goals, Bustk exhibits three novel design aspects. First, Bustk is designed to reap the performance benefits of parallel shadow stacks for real-world applications on modern computer systems while incurring only acceptable memory overhead (often in tens of KBs per thread), which is not as high as expected by the conventional wisdom. More importantly, by using a buddy stack for a thread (with its call stack and shadow stack allocated together at the same time), Bustk not only avoids the need for a shadow stack pointer as before but also makes the parallel shadow stack mechanism well-suited for multi-threading (e.g., on ×86-64).

Second, Bustk uses a new TLS mechanism, STK-TLS, for storing the thread-specific metadata on the buddy stack, that is compatible with existing code (by avoiding program crashes due to register clashes as in REG-TLS) and faster than TCB-TLS (by improving the performance of TCB-TLS). Bustk maintains two metadata sections in dual redundancy on the buddy stack as its TLS, so that both can be accessed and updated from the call stack pointer `rsp`, with only three call-free instructions. This mechanism costs only two physical pages and works seamlessly with ASLR [75] (requiring no special stack alignment).

Finally, Bustk provides integrity protection for the shadow stack by re-randomizing continuously (on the order of milliseconds) its shadowed return addresses with STK-MSR, a new microsecond-level runtime re-randomization technique. This mitigation mechanism introduces a real-time deadline for the attacker (often sitting tens of milliseconds away from the server program being attacked [82]), by obsoleting leaked information and thus making it extremely unlikely for the attacker to hijack return addresses to launch ROP attacks successfully.

As an important caveat, we are aware that network-access delays may not exist for a locally-running exploit script inside a web browser. While Bustk can still be used to protect the return addresses in browsers from being hijacked due to stack buffer overflows, the protection provided by Bustk will be relatively weak. In this case, the attacker may still use, e.g., a locally-running JavaScript program to peak and hijack a return address with a disclosure-aided exploit. In this paper, we have designed Bustk to provide *meaningful security*: we focus mainly on providing strong protection for the return addresses in server programs, but the protection offered for browsers is usable but weaker. In addition, we also aim to introduce a new mechanism for protecting return addresses without relying on information hiding in a large virtual memory space (which is not available on embedded systems without MMU [36, 70]).

Contributions. This paper makes the following contributions:

- We introduce a novel thread-local storage mechanism, STK-TLS, that is both free of compatibility issues (unlike REG-TLS) and efficient (unlike TCB-TLS).
- We introduce a new microsecond-level runtime re-randomization technique, STK-MSR, to mitigate information disclosure attacks by providing integrity protection for the shadow stack with 64-bit entropy.
- We have implemented a hardening tool, BUSTK (STK-TLS + STK-MSR), for protecting return addresses in LLVM (as our backend for translating source code into assembly code) and AFL [87] (for performing assembly-level instrumentation).
- We demonstrate the feasibility of protecting return addresses with parallel shadow stacks for single and multi-threaded server programs. To the best of our knowledge, BUSTK is the first such software hardening tool with continuous runtime re-randomization that can be applied compatibly to real-world single- and multi-threaded server programs.
- We have confirmed experimentally the effectiveness provided by BUSTK in mitigating ROP against server programs, `Nginx1.18` (single-threaded) and `Apache Httpd2.4.46` (multi-threaded). BUSTK-protected servers are highly performant and compatible with unprotected code. In addition, we have also found that the memory overhead incurred per parallel shadow stack is acceptable, costing often tens of KBs only, on average. Finally, BUSTK can provide meaningful security, as it can protect the `Nginx` and `Apache` servers with an adaptive 1-ms re-randomization policy at a low overhead. To evaluate further the performance and compatibility provided by BUSTK against the state of the art, we have also used a number of non-server applications, including `Firefox79.0`, `Python3.8.5`, `Clang/LLVM7.0`, `JDK14`, and the 19 C/C++ benchmarks in `SPEC CPU2006` (as often done in the literature).

2 THREAT MODEL

Just as in previous research on defending against exploits, such as CFI [11, 33, 40, 58, 88, 89] and shadow stacks [12, 21], we assume an attacker with arbitrary read and write primitives. The objective of the attacker is to exploit existing vulnerabilities, e.g., format string in C/C++ programs [15] to launch control-flow hijacking attacks targeting backward edges, i.e., return addresses of functions. Protection for forward edges, i.e., function pointers and virtual table pointers, can be provided orthogonally by defenses such as CFI [11, 33, 40, 58, 88, 89]. How to mitigate data-only attacks, i.e., attacks that do not corrupt code pointers [34, 37], is out of scope for this paper.

In addition, we also assume that standard defenses such as ASLR [75] and DEP [51] are all enabled to further constrain the attacker.

3 METHODOLOGY

We motivate the design of our runtime stack for a thread, called a buddy stack (BUSTK), that consists of a call stack, a parallel shadow stack, and two TLS metadata sections as its TLS, which are all allocated as one single virtual space. We describe our BUSTK design in protecting return addresses bottom-up as follows. We first introduce STK-TLS, a novel stack-based scheme for maintaining thread-specific metadata that allows the metadata to be accessed and updated efficiently from the call stack pointer `rsp` alone (Section 3.1). We then introduce STK-MSR, a microsecond-level runtime re-randomization technique for re-randomizing the return addresses on the shadow stack (Section 3.2). Finally, we combine STK-TLS and STK-MSR together to explain how BUSTK, i.e., our buddy stack, works in protecting return addresses (Section 3.3).

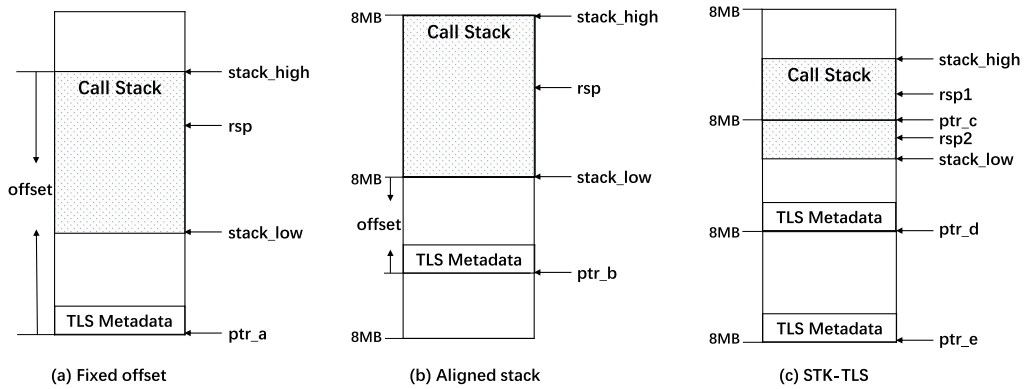

Fig. 2. STK-TLS: Incremental development of Bustk's stack-based TLS (Thread-Local Storage).

3.1 STK-TLS: Bustk's Stack-Based Thread-Local Storage

We introduce STK-TLS in three stages incrementally, as illustrated in Figure 2, for ×86-64. Without loss of generality, we assume a call stack size of 8MB (a commonly-used default in 64-bit Linux) with its two bounds marked by `stack_high` and `stack_low`. A stack position labeled by "8MB" indicates that the position is aligned on an 8MB boundary. We write `TLS Metadata` to refer to all the thread-local information to be maintained by Bustk. As is customary, register `rsp` is the call stack pointer on ×86-64.

We discuss the three solutions below, explaining the incremental development of STK-TLS and highlighting the challenges for designing a stack-based scheme that is both highly performant and compatible with existing code.

3.1.1 Fixed Offset. As shown in Figure 2(a), we store `TLS Metadata` pointed by `ptr_a` at a fixed offset (larger than 8MB) from the call stack. If we know `stack_high`, then `TLS Metadata` can be accessed directly according to `ptr_a = stack_high - offset`, where `offset` is the predefined offset of `TLS Metadata` to `stack_high`.

However, how do we obtain `stack_high` efficiently then? Keeping it in a register as in REG-TLS (Figure 1(a)) offers no compatibility while getting it by calling `pthread_attr_getstack()` would be too expensive as in TCB-TLS (Figure 1(b)). The obvious answer now is to store `stack_high` as part of `TLS Metadata` in **TLS (Thread-Local Storage)**, but we need to know `stack_high` before we can access `TLS Metadata`! Unfortunately, we are now trapped in a chicken-and-egg situation. Thus, "Fixed Offset" does not serve our purposes well.

3.1.2 Aligned Stack. As shown in Figure 2(b), if we further assume that the call stack is aligned exactly on 8MB boundaries, we can then obtain `stack_low` by performing a simple bit-and operation between the call stack pointer `rsp` and a mask (`-0x800000`, i.e., `0xffffffffff800000`): `stack_low = rsp & -0x800000`. As a result, the `TLS Metadata` pointed by `ptr_b` can be simply obtained according to `ptr_b = stack_low - offset`, where `offset` is the predefined offset of `TLS Metadata` to `stack_low`.

"Aligned Stack" is feasible on a 64-bit platform due to the large virtual address space available (and also avoids the need to maintain `stack_low`). However, it does not work with the default ASLR applied for randomizing the base addresses of code and data sections (including the call stack).

3.1.3 STK-TLS. Our stack-based TLS mechanism, as illustrated in Figure 2(c), requires neither the call stack bound information (`stack_high` as in "Fixed Offset") nor any special stack alignment (as in "Aligned Stack"). However, we can still find automatically the required `TLS Metadata` from `rsp`.

There are two cases depending on where `rsp` is currently pointing to: (1) `rsp1`, where `rsp` sits between `stack_high` and `ptr_c`, and (2) `rsp2`, where `rsp` sits between `ptr_c` and `stack_low`. Note that `ptr_c` points to an 8MB-aligned position as shown. As it is hard to know in which of these two cases that `rsp` is at runtime, STK-TLS maintains `TLS Metadata` in dual redundancy, costing only one physical page each, so that the thread-specific metadata can be accessed efficiently from `rsp` alone, no matter whether `rsp` ends up being `rsp1` or `rsp2`. In this mechanism, the low ends of the two `TLS Metadata` sections, which are marked by `ptr_d` and `ptr_e`, respectively, are required to be aligned on 8MB (i.e., the stack size) as shown, with the pointers linking each other, so that finding one enables the other to be found straightaway.

Now, we have a simple and efficient mapping function $\mathscr{M}$ to locate the `TLS Metadata` from `rsp` as follows:

$$\mathscr{M}(rsp) = (rsp \;\&\; -stack\_size) - stack\_size \tag{1}$$

where *stack_size* denotes the call stack size used.

With STK-TLS, we can find the thread-specific information as follows. If `rsp` is `rsp1`, $\mathscr{M}$(rsp) will take us to the `TLS Metadata` pointed to by `ptr_d`, since "`rsp & -stack_size`" gives rise to `ptr_c`. If `rsp` is `rsp2`, $\mathscr{M}$(rsp) will take us to the `TLS Metadata` pointed to by `ptr_e`, since "`rsp & -stack_size`" gives rise to `ptr_d`.

STK-TLS offers compatibility (without using any dedicated register) and is efficient. Figure 3 compares the performance of REG-TLS, TCB-TLS and STK-TLS when a long integer is moved from each type of TLS to a register in PIC, performed inside a loop of N iterations (fully unrolled to ignore any loop-controlling overhead). STK-TLS is 4.43x slower than REG-TLS but 4.26× faster than TCB-TLS, on average. Therefore, for a given shadow stack design (regardless of whether it is parallel or compact), STK-TLS (as adopted in BUSTK proposed in this paper) is expected to be significantly more efficient than TCB-TLS in providing its underlying TLS mechanism.

When illustrating our design in Figure 2, we have assumed a stack size of 8MB. In Section 4.2, we will discuss how STK-TLS can be used to generate hardened code for any power-of-two stack size.

3.2 STK-MSR: BUSTK's Runtime Re-Randomization

When protecting the return addresses in the call stack with a parallel shadow stack, we must enforce the integrity of the shadow stack to mitigate information disclosure attacks [27, 32, 59].

Existing runtime re-randomization techniques [5, 47, 82] operate on concrete code addresses (including function and return addresses). Figure 4 describes their basic idea. Once the function `foo()` is moved from `A` to `B` (which are at a distance of d apart) in code plane at runtime, all the occurrences of its function address saved in data plane (i.e., stack, heap, and global regions) must be tracked and updated from `foo` to `foo+d`. The same must also be done to the return addresses on the call stack. However, this process requires expensive and difficult pointer tracking. For example, RUNTIMEASLR [47] (a state-of-the-art runtime re-randomizer) takes 35 seconds to track the pointers for the `Nginx` web server. Worse still, both false positives and false negatives may exist [5, 47].

Our integrity protection is illustrated in Figure 5. We instrument a protected function `foo()` as follows. At the prologue of `foo()`, we obtain its return address (pushed by its caller) from the call stack, add a random value $\mathscr{D}$ to this return address, and finally, push the randomized return address

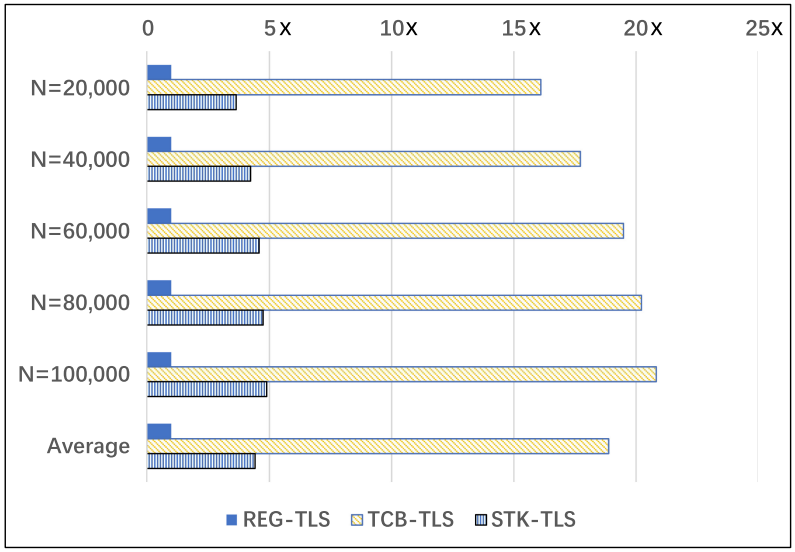

Fig. 3. Comparing STK-TLS with REG-TLS and TCB-TLS in terms of their performance when a long integer is moved from each type of TLS to a register in PIC, performed inside a fully unrolled loop of N iterations.

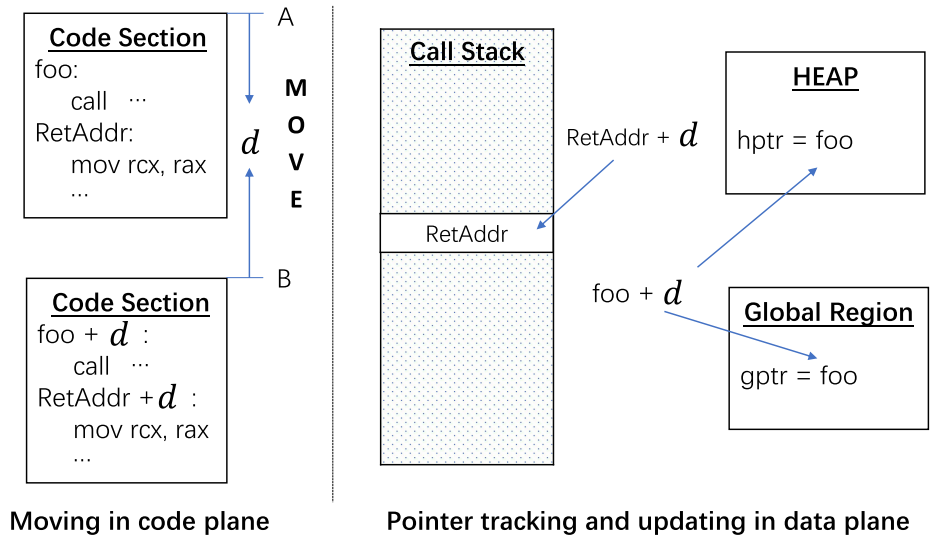

Fig. 4. Runtime re-randomization in code plane.

on the shadow stack. At the epilogue of `foo()`, we obtain this randomized return address from the shadow stack and de-randomize it, i.e., restore the original return address by subtracting $\mathscr{D}$ from the randomized return address. On the left side of Figure 5, when instrumenting code sections, we assume that the return address `RetAddr` on the call stack is read and saved in register `r` and the randomized return address `RandRetAddr` is moved into register `R`.

To provide integrity protection for the shadow stack, we apply a new microsecond-level runtime re-randomization technique, STK-MSR, to re-randomize continuously (on the order of

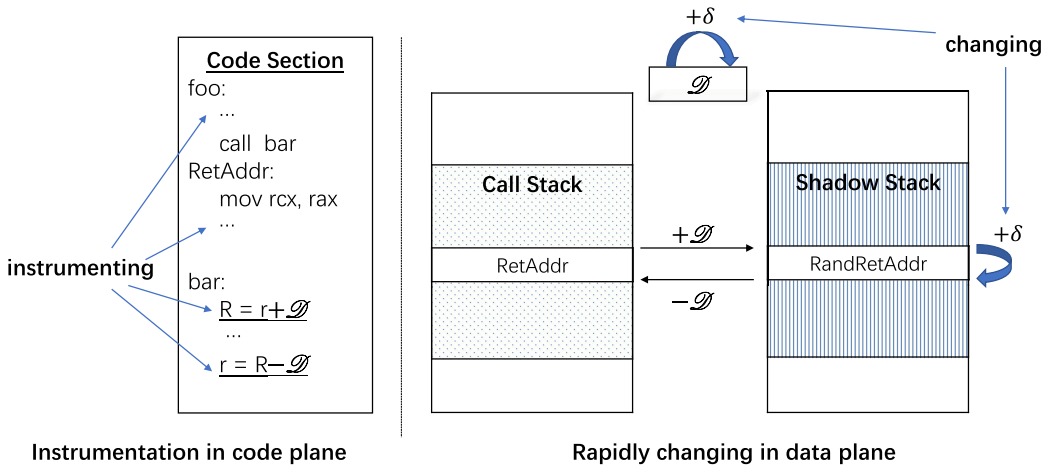

Fig. 5. STK-MSR: BUSTK's runtime re-randomization on the shadow stack.

milliseconds) the return addresses on the shadow stack. As highlighted by "$+\delta$", one round of re-randomization involves incrementing $\mathscr{D}$ and the return addresses on the shadow stack by δ, which is a new random value generated during re-randomization. In a way, δ can also be seen as playing a similar role as d in code plane in Figure 4, except that moving in code plane (Figure 4) has now been converted into rapidly changing in data plane (Figure 5). In addition, δ (consisting of 64 bits) has larger entropy than d (consisting of 47 bits). For a multi-threaded program, $\mathscr{D}$ will be stored as a `TLS Metadata` in dual redundancy based on STK-TLS and δ will be just a local temporary. As randomization techniques provide probabilistic protections against memory exploits, larger entropy is expected to make it harder for the attacker to guess δ (especially for server programs).

STK-MSR is simple yet effective in mitigating ROP attacks. By updating $\mathscr{D}$ timely on the order of milliseconds, we can obsolete any old value of $\mathscr{D}$ leaked to the attacker (sitting often tens of milliseconds away from the server program being attacked) and thus invalidate the return addresses fabricated in a forthcoming ROP payload (built based on an obsolete value of $\mathscr{D}$), as discussed in more detail in Section 3.3 below.

3.3 Buddy Stacks: BUSTK's Runtime Stacks

Our buddy stack layout, BUSTK, is illustrated in Figure 6 by assuming *stack_size* = 8MB. For each thread, we allocate its buddy stack in one single *virtual* address space of size $4 \times$ *stack_size*, which is practically feasible on ×86-64. We examine below its three novel aspects to understand how performance, compatibility, and security are provided, as desired.

3.3.1 Shadow Stack. This is a parallel shadow stack sitting just below the call stack, as each other's buddies, forming BUSTK's ***function-local storage* (FLS)** of size $2 \times$ *stack_size*. In practice, the physical call stack is often small, consuming about tens of KBs (in all real-world applications evaluated in Section 5), and consequently, rendering the physical shadow memory overhead to be small. By allocating a thread's FLS this way, BUSTK not only avoids the shadow stack pointer as before but also makes the parallel shadow stack well-suited for multi-threading (which is not possible traditionally [12]).

3.3.2 STK-TLS. The two `TLS Metadata` sections, which serve as each other's buddies, reside just below the shadow stack as BUSTK's ***thread-local storage* (TLS)** of size $2 \times$ *stack_size*. Their

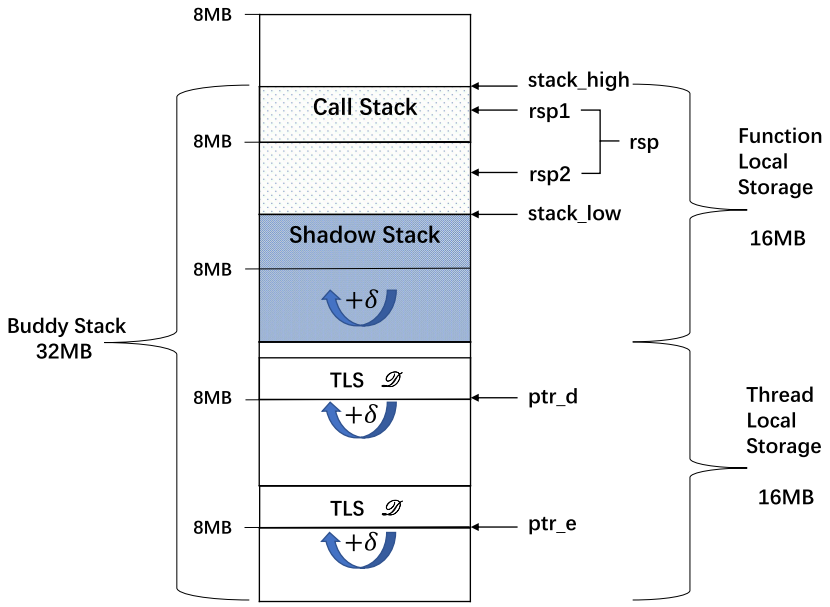

Fig. 6. The buddy stack layout (illustrated with `stack_size` = 8MB).

low ends, which are pointed to by `ptr_d` and `ptr_e`, are aligned on stack size boundaries. Due to the placement of the shadow stack, our mapping function $\mathscr{M}(rsp)$ for finding the `TLS Metadata` is now simply adjusted as:

$$\mathscr{M}(rsp) = (rsp \;\&\; -stack\_size) - 2 \times stack\_size \tag{2}$$

In our current design, the two `TLS Metadata` sections maintain a random value $\mathscr{D}$ in dual redundancy, together with the pointers linking each other, costing one physical page each. In fact, every 8-byte slot in TLS can be seen as a per-thread 64-bit virtual register, if needed. As discussed earlier (Figure 3), STK-TLS offers compatibility unlike REG-TLS (e.g., Shadesmar [12]) and efficiency unlike TCB-TLS, as the `TLS Metadata` can be accessed in three call-free instructions via $\mathscr{M}(rsp)$.

3.3.3 STK-MSR. Bustk provides integrity protection for the shadow stack by re-randomizing its shadowed return addresses continuously (on the order of milliseconds) and efficiently (in tens of microseconds). In our current design, the shadow stack is disclosed yet writable, but $\mathscr{D}$ maintained by STK-TLS is only writable during re-randomization. When STK-MSR happens, Bustk will add a random value δ (indicated by "$+\delta$" in Figure 6) to $\mathscr{D}$ and all the shadowed return addresses.

As a mitigation mechanism, STK-MSR introduces a real-time deadline for an attacker, making it extremely unlikely for the attacker to hijack a return address based on an obsolete value of $\mathscr{D}$ leaked earlier. Suppose the attacker has observed a value of $\mathscr{D}$ as $\mathscr{D}_{\text{old}}$ and would like to change a randomized return address $r+\mathscr{D}_{\text{old}}$ on the shadow stack to $r'+\mathscr{D}_{\text{old}}$, where r' represents a malicious return address in the attacker's ROP payload. However, before this change is made, STK-MSR has already done at least one round of re-randomization, so that $\mathscr{D}$ contains $\mathscr{D}_{\text{new}}$, where $\mathscr{D}_{\text{new}} = \mathscr{D}_{\text{old}}$ is unlikely. Subsequently, zero or more rounds of re-randomization may have happened, so that $\mathscr{D}$ contains $\mathscr{D}_{\text{new}} + \Delta$ and $r' + \mathscr{D}_{\text{old}}$ becomes $r' + \mathscr{D}_{\text{old}} + \Delta$. When $r' + \mathscr{D}_{\text{old}} + \Delta$ is fetched for execution, the de-randomized address is $r' + \mathscr{D}_{\text{old}} - \mathscr{D}_{\text{new}}$, which is extremely unlikely to be the malicious return address, r', as intended. Thus, Bustk protects the shadow stack probabilistically with 64-bit entropy (as $\mathscr{D}$ is a 64-bit random value) for server programs and locally-running programs without

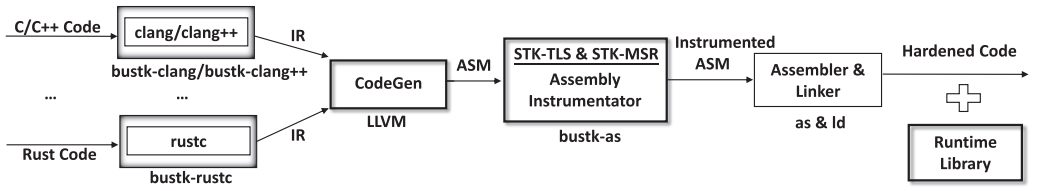

Fig. 7. The workflow of Bustk for protecting the return addresses in a program with the program instrumented at its assembly code.

a script runtime. Otherwise, the protection offered by Bustk is weaker (as already acknowledged in Section 1).

4 BUSTK: DESIGN AND IMPLEMENTATION

Bustk consists of STK-TLS and STK-MSR. Both are loosely-coupled, with each also being applicable to many other scenarios. For example, STK-TLS can be used for providing highly performant TLS where compatibility is required. STK-MSR can be used to provide integrity protection for compact shadow stacks where a register can be dedicated to maintain the shadow stack pointer as in some embedded systems [3] (Section 5.7). In this paper, STK-MSR can also be understood as an important application of STK-TLS to demonstrate its efficiency and compatibility for multi-threaded programs.

We describe the design and implementation of Bustk, as shown in Figure 7. Bustk uses LLVM as its backend for translating source code into assembly code and AFL [8, 87] to produce instrumented assembly code. To provide compatibility with a range of LLVM backends, Bustk is designed as an assembly instrumentator, `bustk-as`. The assembly code generated by an LLVM backend will be intercepted by our compiler wrappers (Section 4.1), then instrumented by `bustk-as` to support STK-TLS and STK-MSR (Section 4.2), and finally, passed to the native assembler and linker to generate a hardened binary for supporting continuous runtime re-randomization (Section 4.3), which is linked with our simple Bustk runtime library (Section 4.4).

4.1 Compiler Wrappers

Bustk can offer protection for programs written in a variety of programming languages where their return addresses need to be protected. In our evaluation, however, we have focused mainly on protecting the return addresses in C/C++ programs, as all the benchmarks and applications considered are coded in C/C++, except that `Firefox` is partly coded in Rust.

Given a program written in C/C++/Rust as input, Bustk will generate a hardened binary protected by continuous runtime re-randomization. For users, our current compiler wrappers, `bustk-clang`, `bustk-clang++` and `bustk-rustc`, can be easily used as a drop-in replacement for `clang`, `clang++`, and `rustc`, respectively. At the front end of a compiler tool chain `T` for a given language, Bustk will act as a compiler wrapper, `bustk-T`. In essence, `bustk-T` will forward all the compiler options to `T` for `T` to do its own business as usual, except that `T` will terminate after having generated assembly code (as an external assembler, i.e., `bustk-as` instead of its integrated assembler will be used).

Figure 8 shows how `bustk-rustc` looks like. In line 2, a large enough memory space is allocated for the command-line options to be passed to `rustc`. The first one is set as the path of `rustc` (line 4) and the others are copied verbatim from `argv` (lines 5-8). Some additional options can be added as illustrated in lines 9-10, with `no-integrated-as` requesting `rustc`'s integrated assembler to be disabled. Finally, `rustc` is executed by invoking `execvp` (line 12).

```
01 int main(int argc, char **argv){
02    char **params = malloc(N);
03    int cnt = 1;
04    params[0] = "/usr/bin/rustc";
05    while (--argc) {
06        char *cur = *(++argv);
07        params[cnt++] = cur;
08    }
09    params[cnt++] = "-C";
10    params[cnt++] = "no-integrated-as";
11    params[cnt] = NULL;
12    execvp(params[0], params);
13    FATAL_ERR("%s", params[0]);
14    return 0;
15 }
```

Fig. 8. The compiler wrapper for `rustc`.

```
01 test:
02  # Prologue: RandRetAddr = RetAddr + 𝒟
03  400770:     49 89 e3                 mov    %rsp,%r11
04  400773:     49 81 e3 00 00 80 ff     and    $-0x800000,%r11
05  40077a:     4d 8b 9b 00 00 00 ff     mov    -0x1000000(%r11),%r11
06  400781:     4c 03 1c 24              add    (%rsp),%r11
07  400785:     4c 89 9c 24 00 00 80 ff  mov    %r11,-0x800000(%rsp)

    ...
08  # Epilogue: RetAddr = RandRetAddr - 𝒟
09  4007b7:     49 89 e2                 mov    %rsp,%r10
10  4007ba:     49 81 e2 00 00 80 ff     and    $-0x800000,%r10
11  4007c1:     4c 8b 9c 24 00 00 80 ff  mov    -0x800000(%rsp),%r11
12  4007c9:     4d 2b 9a 00 00 00 ff     sub    -0x1000000(%r10),%r11
13  4007d0:     48 83 c4 08              add    $0x8,%rsp
14  4007d4:     41 ff e3                 jmpq   *%r11
```

Fig. 9. Assembly instrumentation of a protected function `test()` for supporting STK-TLS and STK-MSR on ×86-64.

4.2 Assembly Code Instrumentation

To protect a program with a buddy stack, its instrumented code will depend on *stack_size*. We first describe a solution with *stack_size* being hard-coded, which is faster of the two and expected to be already practically deployable. We then describe another more expensive solution where *stack_size* is stored in a global variable, avoiding a hard-coded stack size.

Figure 9 shows how a protected function `test()` is instrumented by adding a prologue and an epilogue to support STK-TLS and STK-MSR by assuming *stack_size* = 8MB on ×86-64. The two registers, `r10` and `r11`, used comply with the requirements of the System V ABI [50]. Note that $\mathscr{M}$(`rsp`) in Equation (2) can be reached in three call-free instructions.

The prologue serves to store the return address RetAddr of test() pointed by rsp as RandRetAddr = RetAddr + $\mathscr{D}$ on the shadow stack, as indicated in line 2. According to STK-TLS, the three instructions are used to fetch $\mathscr{D}$ based on $\mathscr{M}$(rsp) given in Equation (2) (lines 3-5): rsp is first copied to r11 (line 3) and then aligned on 8MB (line 4), with $\mathscr{D}$ being finally fetched into r11 (line 5). According to STK-MSR (lines 6-7), RetAddr is first fetched from the call stack and then added to r11 (line 6), and subsequently, RetAddr + $\mathscr{D}$ is pushed to the shadow stack (line 7).

The epilogue retrieves the randomized return address of test(), RandRetAddr, from the shadow stack and then redirect program execution to RetAddr = RandRetAddr − $\mathscr{D}$ (after de-randomization), as indicated in line 8. In this case, -0x800000(%rsp), which is copied to r11, in line 11, contains RandRetAddr, and -0x1000000(%r10) in line 12 contains $\mathscr{D}$ (computed in lines 9, 10, and 12 according to $\mathscr{M}$(rsp)). After line 12, r11 thus contains RetAddr. In line 13, rsp is adjusted to skip RetAddr on the call stack. Finally, test() returns via an indirect jump in line 14.

In BUSTK, we have opted not to validate the return address on the shadow stack against the corresponding return address on the call stack at the epilogue of a protected function. In the case when the shadow stack has been exposed, such a validation will be meaningless, since the attacker should have already corrupted the two return addresses at the same time (in one payload) with arbitrary writes, as discussed in our threat model and demonstrated in [96] by exploiting some format string vulnerabilities in one payload. On the other hand, if the attacker has not managed to locate the shadow stack, they are usually shrewd enough not to corrupt the return address on the call stack first at the risk of being caught without being able to achieve their goals.

In both the prologue and epilogue, the underlined masks and offsets are dependent on *stack_size*. These hard-coded immediates may seemingly be problematic when BUSTK is adopted and deployed. Suppose two shared libraries are hardened with different call stack sizes at different times. If both are used by the same program, we can actually use a simple binary rewriter [26, 79, 83] (in Python) to adjust these hard-coded masks and offsets by pattern matching for a given stack size. For example, we find that this works well when we rewrite all the shared libraries and the main executable for a BUSTK-protected Firefox browser. Alternatively, we can also avoid binary rewriting by hard-coding stack_size as 8MB (as demonstrated in Figure 9), as 8MB is large enough for nearly all practical purposes on ×86-64 (as evaluated in Section 5).

We can also avoid hard-coding stack_size by keeping it in a global variable and generate the prologue and epilogue for a function accordingly. This scheme is slightly more expensive in both performance and code size overheads incurred. Figure 10 describes Bustk.Global, a version of BUSTK, for instrumenting a protected function test() for supporting STK-TLS and STK-MSR by encoding *stack_size* in a global variable. In lines 4 and 11, −*stack_size* is saved in a global variable, named .BUDDY.STK.MASK. Now, there is not any hard-coded immediate in the instrumented assembly code as in Figure 9. Thus, when a program starts, *stack_size* can be adjusted as needed and the data page containing the global variable .BUDDY.STK.MASK is then set as read-only. The assembly code in Figure 10 (for Bustk.Global) can be understood similarly as that in Figure 9 (for BUSTK). In both cases, $\mathscr{M}$(rsp) given in Equation (2) is used to access the TLS Metadata provided by STK-TLS and implement the shadow stack integrity protection provided by STK-MSR.

BUSTK requires five instructions instrumented at the prologue and six at the epilogue. In contrast, Bustk.Global requires more memory accesses (in lines 4 and 11) and more instructions (six at the prologue and eight at the epilogue). Therefore, we recommend BUSTK to be adopted, despite its use of hard-coded stack sizes in hardened binaries.

```
01 test:
02  # Prologue: RandRetAddr = RetAddr + 𝒟
03  4007b0: 49 89 e3                  mov    %rsp,%r11
04  4007b3: 4c 8b 14 25 00 30 60 00   mov    .BUDDY.STK.MASK,%r10
05  4007bb: 4d 21 d3                  and    %r10,%r11
06  4007be: 4f 8b 1c 53               mov    (%r11,%r10,2),%r11
07  4007c2: 4c 03 1c 24               add    (%rsp),%r11
08  4007c6: 4e 89 1c 14               mov    %r11,(%rsp,%r10,1)
    ...
09  # Epilogue: RetAddr = RandRetAddr - 𝒟
10  4007f4: 49 89 e3                  mov    %rsp,%r11
11  4007f7: 4c 8b 14 25 00 30 60 00   mov    .BUDDY.STK.MASK,%r10
12  4007ff: 4d 21 d3                  and    %r10,%r11
13  400802: 4f 8b 1c 53               mov    (%r11,%r10,2),%r11
14  400806: 4e 8b 14 14               mov    (%rsp,%r10,1),%r10
15  40080a: 4d 29 da                  sub    %r11,%r10
16  40080d: 48 83 c4 08               add    $0x8,%rsp
17  400811: 41 ff e2                  jmpq   *%r10
```

Fig. 10. Assembly instrumentation of a protected function `test()` for supporting STK-TLS and STK-MSR by encoding *stack_size* in a global variable in a version of Bustk, called `Bustk.Global`.

4.3 Runtime Re-Randomization

As this work focuses on protecting server programs, STK-MSR can be activated in many ways: at a preset period [82] and/or an I/O system call [5] (among others). When and where to apply STK-MSR is a policy issue, depending on the computing environment at hand.

In this paper, we focus on developing a new microsecond-level re-randomization mechanism while leaving the design of a re-randomization policy as an orthogonal issue. Currently, we adopt an adaptive policy by hooking the I/O functions (via `LD_PRELOAD`) and polling on the ×86 time stamp counter in these functions to check if STK-MSR should be activated (again). This simple policy appears to be adequate enough by both protecting server programs (where I/O is performed frequently) and stress-testing STK-MSR to demonstrate its lightweightness when invoked even once every 1 ms. Thus, we write `Bustk.n-ms` for such an adaptive policy, by which STK-MSR is activated in a hooked I/O function if STK-MSR has not been done in the past `n` milliseconds.

Consider the `Nginx` server protected by `Bustk.1-ms` (with STK-MSR happening at a frequency of at most 1,000 times per second). The server uses `recv()`, which is thus hooked, to read data from remote clients. If the data arrives at a higher frequency, STK-MSR will only be activated 1,000 times per second. Such a re-randomization frequency is fast enough to make obsolete a previously leaked value of $\mathscr{D}$, as servers are reported to often sit tens of milliseconds away from attacker machines [82]. On the other hand, if the client data arrives at a lower frequency, then STK-MSR may be triggered at every `recv()` call, but in under 1,000 times per second. Thus, our default policy applies re-randomization with an adaptive frequency/period per thread. For single-threaded servers like `Nginx`, `Bustk.1-ms` suffices to invalidate a previously leaked value of $\mathscr{D}$, as STK-MSR happens every 1 ms or before every `recv()` call. For multi-threaded servers, where cross-thread stack-smashing attacks [85] are possible, a more sophisticated re-randomization policy can be used, as discussed in Section 5.7, since STK-MSR is lightweight.

Figure 11 gives `runtime_rerandomize()`, our algorithm for performing fast SIMD-based runtime re-randomization on ×86-64, where `tls` points to one `TLS Metadata` section according to

```
01 void runtime_rerandomize (TLS_Metadata *tls){
02    long δ = random();
03    long 𝒟 = tls->𝒟;
04    TLS_Metadata *buddy = tls->buddy;
05    mprotect(tls, PAGE_SIZE, PROT_WRITE | PROT_READ);
06    tls->𝒟 += δ;
07    mprotect(tls, PAGE_SIZE, PROT_READ);
08    mprotect(buddy, PAGE_SIZE, PROT_WRITE | PROT_READ);
09    buddy->𝒟 += δ;
10    mprotect(buddy, PAGE_SIZE, PROT_READ);
11    SIMD_add( (long *)(GET_RSP() - CALL_STK_SIZE), (long *)(tls->stack_top - CALL_STK_SIZE), δ);
12    assert(𝒟 + δ == tls->𝒟 && 𝒟 + δ == buddy->𝒟);
13 }
14 inline void SIMD_add (long *ss_ptr, long *ss_top, long δ){
15    // 32-byte aligned
16    ss_ptr = (long *) (((long) ss_ptr) & (-32))
17    // declare three 256-bit registers
18    m256i A, B, C;
19    long __attribute((aligned(32))) xs[] = {δ, δ, δ, δ};
20    B = _mm256_load_si256((__m256i *)xs);
21    while(ss_ptr < ss_top){
22        A = _mm256_load_si256((__m256i *) ss_ptr);
23        C = _mm256_add_epi64(A, B);
24        _mm256_store_si256((__m256i *) ss_ptr, C);
25        // 4 long integers
26        ss_ptr += 4;
27    }
28 }
```

Fig. 11. Runtime re-randomization, STK-MSR, for providing integrity protection on ×86-64.

$\mathscr{M}$(rsp). Its buddy TLS Metadata section is obtained in line 4. In Bustk, the two physical pages containing the TLS Metadata are writable during re-randomization only. The parallel shadow stack, marked by GET_RSP() - CALL_STK_SIZE and tls->stack_top - CALL_STK_SIZE, is always readable and writable, where GET_RSP() returns the value of rsp, tls->stack_top points to the top of the buddy, i.e., call stack, and CALL_STK_SIZE represents stack_size.

We re-randomize all the shadowed return addresses by incrementing each with a randomly generated value δ (line 2). We first update $\mathscr{D}$ to $\mathscr{D}+\delta$ (lines 5-10). We then update the return addresses on the shadow stack by calling SIMD_add() (line 11). Instead of modifying each shadowed return address r to $r+\delta$ by using the unwind information [82, 96], we have opted to a simpler yet faster technique that increments all the long integers on the shadow stack by δ with a four-way SIMD instruction. As the physical call stack is small, costing about tens of KBs on average (Section 5), we can perform one round of re-randomization on the order of microseconds.

When SIMD_add() is called, ss_ptr represents the shadow stack pointer, ss_top points to the top of the shadow stack, and δ represents the amount to be added to each shadowed return address. In line 16, ss_ptr is 32-byte aligned. In line 19, δ is duplicated and packed into a 256-bit array (of 4 long integers), which is then loaded into a 256-bit register, B. In lines 21-27, a loop is used to

ALGORITHM 1: *GetCallStackInfo*()

```
   Input  : An executable binary
   Output : The largest call stack depth and size during an execution
1  Procedure GetCallStackInfo()
2  max_depth ← 0;
3  max_size ← 0;
4  depth ← 0;
5  stack_high ← getStackPtr();
6  foreach ins ∈ AllInstructions do
7      if isReturn(ins) then
8          if depth > 0 then
9              depth ← depth − 1;
10     else if isCall(ins) then
11         depth ← depth + 1;
12         if depth > max_depth then
13             max_depth ← depth;
14     else if isStackPtrChanged(ins) then
15         stack_size ← stack_high − getStackPtr();
16         if stack_size > max_size then
17             max_size ← stack_size;
```

perform a 4-way 256-bit addition of δ to the long integers in [`ss_ptr`, `ss_top`], assisted by two more 256-bit registers, `A` and `C`.

4.4 Runtime Library

Our runtime library is simple. When a thread starts, the two `TLS Metadata` sections (as shown in Figure 6) are initialized in dual redundancy, with $\mathscr{D}$ being assigned a random value and the top of the buddy, i.e., call stack (i.e., `tls->stack_top` in line 11 of Figure 11) being recorded. In addition, both sections are set up as each other's buddies (line 4 of Figure 11). A per-thread flag is kept to mark whether a thread is doing its re-randomization. Finally, the current time (measured in terms of CPU cycles since reset in the ×86 time stamp counter) is recorded in order to support continuous runtime re-randomization in the thread.

4.5 Collecting Call-Stack Usage Information

We have developed a simple instrumentation tool in terms of Algorithm 1, based on Intel Pin [49], to collect the call stack usage information, i.e., stack size (denoted *max_size*) and stack depth (denoted *max_depth*), in Table 8, for one particular execution of a given thread.

The basic idea is simple. The initial call stack pointer `rsp` is saved in *stack_high* in line 5. Every executed instruction `ins` is instrumented (line 6). If *ins* is a return, *depth* decreases by 1 (lines 7-9). If *ins* is a call, then *depth* increases by 1 and *max_depth* is updated if needed (lines 10-13). If *ins* may change `rsp`, *max_size* is adjusted if needed (lines 14-17).

5 EVALUATION

Our evaluation aims to show that Bustk offers compatibility, performance, and meaningful security while incurring some small memory and code size overheads in protecting backward edges, i.e., return addresses in server programs. We demonstrate this by comparing Bustk with Shadesmar [12], a state-of-the-art REG-TLS-based tool for protecting return addresses with compact

shadow stacks. In particular, we have selected SHADESMAR's most efficient REG-TLS-based scheme, SHADOW_STACK_REG (without using MPX [35] or MPK [63] for integrity protection).

Our platform consists of a 3.20 GHz Intel(R) Core(TM) i5-6500 CPU with 16 GB memory, running the 64-bit Ubuntu 18.04 OS. To compare BUSTK with SHADESMAR [12] fairly, we use LLVM 7.0, the same compiler framework where SHADESMAR is implemented, for building our programs. To evaluate BUSTK against SHADESMAR in terms of three metrics, compatibility, performance, and security, we consider two popular web servers, `Nginx` (single-threaded) and `Apache Httpd` (multi-threaded). To compare both further (especially in terms of performance and compatibility), we have also used non-server applications, `Firefox`, `Python`, `Clang/LLVM`, `JDK`, and the 19 C/C++ benchmarks in `SPEC CPU2006` (as in [12]), where `Firefox`, `Python`, and `JDK` are multi-threaded, as is also done in the literature. For each SPEC CPU2006 benchmark, `Nginx` and `Apache Httpd`, the optimization flag used is "-O3". For other applications, the default build setting that comes with their source code is used. For example, "-O2" and "-O3" are used for compiling different modules of `Firefox`. The measurement for each metric is obtained as the average of five runs (unless otherwise indicated). The call stack size used for each BUSTK-protected application is is 8MB.

5.1 Performance

We show that BUSTK is highly performant, by comparing with SHADESMAR [12] (with the MPX/MPK-protection turned off for the shadow stack, i.e., SHADOW_STACK_REG). As reported in [12], the average performance overheads of MPX and MPK are 12.12% and 61.18%, respectively, and are thus not acceptable for a deployed mechanism. Even if the MPX/MPK-related performance overheads in SHADESMAR are ignored in our evaluation, BUSTK still has comparable performance overheads as SHADESMAR. In particular, BUSTK and `Bustk.1-ms` (with an adaptive 1-ms re-randomization policy) exhibit performance overheads that are small enough for a practical deployment.

5.1.1 Web Servers. To evaluate BUSTK in server settings, we use a pair of 4-core machines (of the same configuration described at the beginning of Section 5) on a dedicated **local area network (LAN)**, with one functioning as the server and the other as the client. Two server programs, `Nginx` (single-threaded) and `Apache Httpd` (multi-threaded), are evaluated separately. To minimize noise, no other machines are also allowed to be on the LAN. On the server machine, `Nginx` and `Apache` are configured to have four worker processes (same as the number of CPU cores available). On the client machine, the standard `ApacheBench` tool is used to send HTTP requests to vanilla, SHADESMAR- and BUSTK-protected `Nginx/Apache` servers. Tables 1 and 2 give the results. For each mitigation tool, we measure its performance overhead as the percentage of requests per second (throughput) reduced. Each measurement is reported as the average of five 120-second runs, with a range of simultaneous connections selected for the concurrency parameter c of `ApacheBench`.

Table 1 gives the performance overheads of SHADESMAR- and BUSTK-protected `Nginx` servers. When $c \leqslant 4$, `Bustk.1-ms` has slightly higher overheads. For both tools, no overheads are observable when $c \geqslant 64$. With $c = 64$, `Nginx` represents an extremely hot website, serving 17,247.43 requests per second, i.e., about 1.49 billion requests per day.

Table 2 gives the performance overheads of the BUSTK-protected `Apache` server. Due to register clashes, the SHADESMAR-protected `Apache` server crashes (as discussed in Section 5.2). According to Table 2, `Bustk.1-ms` exhibits slightly higher performance overheads in (multi-threaded) `Apache` than (single-threaded) `Nginx` under the same setting, implying that multi-threading can induce extra overheads. When $c \geqslant 64$ (i.e., when IO is intensive, reaching and exceeding 16,939.78 requests per second), `Bustk.1-ms` also has no observable impact on the performance of `Apache`.

Table 1. Performance Overhead of SHADESMAR and BUSTK for Nginx

Concurrency	Vanilla Nginx	SHADESMAR	BUSTK.1-ms
c	Requests/Second	Overhead	
1	809.51	0.70%	1.22%
2	1,475.47	0.18%	2.59%
4	3,129.77	0.64%	1.51%
8	6,280.97	−0.43%	0.50%
16	11,511.90	0.00%	−0.21%
32	17,114.67	0.14%	0.22%
64	17,247.43	0.00%	0.00%

For SHADESMAR, its MPX-/MPK- protection for the shadow stack (with an average overhead of 12.12% (MPX) and 61.18% (MPK) [12]) is turned off.

Table 2. Performance Overheads of SHADESMAR and BUSTK for Apache

Concurrency	Vanilla Apache	SHADESMAR	BUSTK.1-ms
c	Requests/Second	Overhead	
1	682.58	N/A	3.74%
2	1,266.99	N/A	3.42%
4	2,497.02	N/A	−0.30%
8	5,197.14	N/A	3.83%
16	11,993.15	N/A	0.67%
32	16,933.47	N/A	0.11%
64	16,939.78	N/A	0.00%

The numbers for SHADESMAR are not available as SHADESMAR can build Apache but fails to run it due to register clashes (with unprotected libraries).

In Tables 1 and 2, the negative overheads are due to system jitter (such as uncertainty of thread scheduling and timing of incoming requests). BUSTK can re-randomize Nginx and Apache on the order of microseconds, as their maximum shadow/call stack sizes are only 9,488 bytes and 35,376 bytes, respectively (reported in Table 8). In fact, BUSTK can re-randomize a shadow stack of size 1MB, the maximum reported in Table 8 and discussed below, in 34 microseconds (Table 4). For smaller shadow stacks (of less than 64KBs), a considerable part of BUSTK's re-randomization overhead is spent on the four system calls to `mprotect()` (Figure 11), in about 6 μs per randomization (Table 4). On newer Intel platforms where MPK [63] is available, this cost can be further reduced [63]. It is also worth emphasizing that when a re-randomization phase occurs, the shadow/call stack sizes are often less than their maximum shadow/call stack sizes given in Table 8.

SHUFFLER [82] can re-randomize a protected program by shuffling the address space layout at runtime, thus making invalid the ROP gadgets collected by an attacker. As reported in [82], SHUFFLER takes about 15 milliseconds to perform one round of re-randomization (shuffling) for Nginx, leading to a performance slowdown of about 15% with a shuffle period of 100 milliseconds and a performance slowdown of 30% with a shuffle period of 50 milliseconds. To reduce performance overhead, shuffling must be done asynchronously in shuffler threads by making use of spare CPU cycles on idle cores. In contrast, BUSTK is substantially more lightweight (by performing one round of re-randomization at the microsecond level), as it re-randomizes only the shadowed return addresses. For a BUSTK-protected server, once the attacker has collected enough ROP gadget addresses by exploiting information disclosure, the remaining obstacle to overcome is to peek $\mathscr{D}$

Table 3. Performance Overheads of SHADESMAR and BUSTK for the (single-threaded) SPEC Benchmarks

Benchmark	SHADESMAR	BUSTK	BUSTK.1-ms
400.perlbench	11.56%	8.00%	10.22%
401.bzip2	3.25%	1.63%	3.52%
403.gcc	10.55%	6.03%	8.54%
429.mcf	−0.96%	0.00%	0.32%
433.milc	2.56%	1.86%	2.33%
444.namd	1.61%	0.00%	1.61%
445.gobmk	9.57%	10.72%	13.04%
447.dealII	**N/A**	**4.63%**	**6.94%**
450.soplex	1.94%	0.97%	2.91%
453.povray	**N/A**	**17.82%**	**19.80%**
456.hmmer	8.10%	0.40%	2.02%
458.sjeng	7.76%	8.91%	10.34%
462.libquantum	2.68%	1.53%	2.30%
464.h264ref	6.61%	8.91%	8.62%
470.lbm	2.51%	0.50%	2.51%
471.omnetpp	**N/A**	**5.18%**	**6.47%**
473.astar	2.73%	3.94%	5.76%
482.sphinx3	2.70%	3.51%	5.68%
483.xalancbmk	**N/A**	**9.64%**	**12.65%**
Average	4.88%	4.96% (3.80%)	6.61% (5.32%)

For SHADESMAR, its MPX-/MPK- protection for the shadow stack (with an average overhead of 12.12% (MPX) and 61.18% (MPK) [12]) is turned off. The four SHADESMAR-protected benchmarks highlighted in **bold** crash due to incompatibility with unprotected code (Section 5.2).

sitting on the buddy stack (Figure 6). Thus, a re-randomization period at tens of milliseconds (say, 50 milliseconds) is not fast enough for BUSTK, considering that a network delay is reported to be often in the order of tens of milliseconds. Unlike SHUFFLER, BUSTK must therefore adopt a smaller re-randomization period (1 millisecond by default in our current implementation).

A runtime re-randomization defense that is effective in mitigating information disclosure attacks must be efficient (performant). According to our evaluation, BUSTK can conduct one round of re-randomization in tens of microseconds (Table 4), making it feasible for `Bustk.1-ms` to re-randomize the `Ngnix` and `Apache` servers at a low overhead (Tables 1 and 2). We believe that an adaptive 1-ms re-randomization policy is fast enough for BUSTK-protected servers to mitigate ROP attacks by making $\mathscr{D}$ leaked earlier obsolete before it is used.

5.1.2 SPEC Benchmarks. To evaluate the performance overheads of BUSTK and SHADESMAR further, Table 3 compares BUSTK (with STK-MSR off) and `Bustk.1-ms` (with STK-MSR on) against SHADESMAR [12] for the 19 (single-threaded) C/C++ SPEC CPU2006 benchmarks (used also in SHADESMAR [12]). The four benchmarks highlighted in **bold**, once protected by SHADESMAR, crash due to incompatibility with unprotected code (as discussed in Section 5.2). In terms of average performance overheads added to the baseline (without any protection), SHADESMAR has 4.88% (even when its MPX-/MPK- protection is off). BUSTK gives rise to 4.96% (3.80%) when the four benchmarks in **bold** are included (excluded). The overhead numbers for `Bustk.1-ms` are slightly higher, reaching 6.61% (5.32%). These results for (single-threaded) SPEC programs demonstrate

Table 4. The Time Spent by Bustk's `runtime_rerandomize()` in Figure 11 under Different Call Stack Sizes

Call Stack Size (KBs)	w/ mprotect (μs)	w/o mprotect (μs)
64	8	1
128	9	3
256	12	6
512	20	14
1,024	34	27
2,048	61	54

Table 5. Scores of the Vanilla and Bustk.1-ms for `Firefox79.0`

	Vanilla	Bustk.1-ms	Overhead
Octane2.0	26,524	25,220	5.17%
JetStream2	72.940	67.013	8.84%

further that Bustk can provide high performance while offering also compatibility (as discussed in Section 5.2).

To see why `Bustk.1-ms` is only slightly more expensive than Bustk, Table 4 analyzes the performance of `runtime_rerandomize()` in Figure 11 for implementing STK-MSR under a range of call stack sizes in [64KB, 2048KB]. As will be discussed in Section 5.4 below, the call stack (or its shadow stack) is about tens of KBs deep on average, hardly exceeding 1MB. To reduce jitter and noise, the time in each measurement is the average of 100,000 runs. As shown in Figure 11, we currently make the two pages containing $\mathscr{D}$ writable only when it is re-randomized, requiring a total of four calls to `mprotect()` at about 6 μs. Without these four calls, the time spent by `runtime_rerandomize()` is nearly proportional to the call stack size. On platforms that support Intel's MPK [63], the new `wrpkru` instruction can be a more efficient alternative [63], as it works by assigning appropriate permissions to the two pages containing $\mathscr{D}$. However, we do not also need to use MPK to protect the shadow stack itself (at the prologue and epilogue of each protected function), thus avoiding its excessively high overheads reported [12]. Currently, Bustk protects the shadow stack probabilistically by STK-MSR with 64-bit entropy.

5.1.3 Browsers. We have benchmarked the performance of `Bustk.1-ms`-protected `Firefox79.0` with `Octane2.0` [24] and `JetStream2` [25]. The former is used for measuring JavaScript engine's performance while the latter is for measuring the performance of JavaScript and WebAssembly operations. `JetStream2` has 64 subtests. For its first subtest (WSL, an implementation of a GPU shading language), we always choose to wait rather than stop when the message "A web page is slowing down your browser. What would you like to do?" pops up. Again, we cannot compare Bustk with Shadesmar as Shadesmar crashes for `Firefox79.0` due to incompatibility (Section 5.2).

Table 5 gives the results. For `Octane2.0`, vanilla `Firefox79.0` has scored 26,524 while `Bustk.1-ms`-protected `Firefox79.0` has scored 25,220, yielding a performance overhead of 5.17% for `Bustk.1-ms`. For `JetStream2`, the scores of vanilla and `Bustk.1-ms`-protected `Firefox79.0` are 72.940 and 67.013, respectively, yielding a slightly higher performance overhead of 8.84%.

5.2 Compatibility

Bustk offers compatibility (due to STK-TLS) but Shadesmar does not (as it is REG-TLS-based) due to the program crashes caused by register clashes illustrated in Figure 1. Table 6 confirms this

Table 6. Compatibility Provided by SHADESMAR and BUSTK in Building/Running Open-source Projects

Benchmark	SHADESMAR		BUSTK	
	Build	Run	Build	Run
Nginx1.18	✓	✓	✓	✓
Apache Httpd2.4.46	✓	×	✓	✓
Python3.8.5	×		✓	✓
JDK14	×		✓	✓
Firefox79.0	×		✓	✓
Clang/LLVM7.0	×		✓	✓

Table 7. The Number of Occurrences of `xmm15` in `Firefox` and its 20 Shared Libraries when Compiled by Vanilla LLVM and the Modified LLVM (with `xmm15` Treated as a Reserved Register)

Program	LLVM 7.0	
	VANILLA	MODIFIED
Firefox	12	0
libsoftokn3.so	72	0
libssl3.so	18	0
libmozavutil.so	3	0
libfreeblpriv3.so	365	61
libxul.so	14,781	2,491
libmozavcodec.so	4,948	4,218
Other 14 shared libraries	0	0

by using both mitigation tools to build the same six important open-source projects listed. For the two servers, SHADESMAR can build and run the `Nginx` server only. For the other four non-server applications, SHADESMAR fails in automated testing during the build. In contrast, BUSTK can build and run all the six projects successfully, ensuring compatibility with existing code.

Let us analyze `Firefox` to see why a REG-TLS-based approach may sacrifice compatibility. As SHADESMAR crashes at an early build stage, we turn to vanilla LLVM to find out how it has used `r15` (the register reserved by SHADESMAR for maintaining its shadow stack pointer). It turns out that `r15` is used in `Firefox` and 17 out of its 20 shared libraries generated by vanilla LLVM. Perhaps, this register-clashing problem can be avoided if a rarely-used register [44], say, `xmm15` is reserved instead? Unfortunately, this is not the case as shown in Table 7 (with 14 shared libraries combined into the last row). For vanilla LLVM, `xmm15` is used in `Firefox` and six shared libraries. For the modified LLVM (with `xmm15` reserved), `xmm15` is now less frequently used, but appears still in three shared libraries, indicating that `Firefox` relies on some libraries in which `xmm15` cannot be reserved as expected. To check why `xmm15` still exists in the three shared libraries, we have manually examined `libfreeblpriv3.so` and found that it relies on two hand-written assembly source files, `intel-aes.s` and `intel-gcm.s`, where `xmm15` is hard-coded in the assembly code. In `Firefox`, there are 584 assembly files, with 278 file names ending with ".s" (or ".S") and 306 file names ending with ".asm". This shows that hand-crafted assembly code is another source of register clashes for a REG-TLS-based approach like SHADESMAR. This issue can only become worse for browsers if closed-source plugins (often released as shared libraries) are also considered [2, 52].

```
01 // ld-linux-x86-64.so.2
02 void _dl_init (struct link_map *main_map, ...){
03   # push   %r15
04   # mov    %rdi,%r15
05   ⋯
06   while (i-- > 0){
07     # mov    0x3b8(%r15),%rdx
08     # mov    %ebx,%eax
09     # mov    (%rdx,%rax,8),%r14
10     call_init (main_map->l_initfini[i], ...);
11   }
12   # pop    %r15
13   # retq
14 }
15 // stack trace
16 # _GLOBAL__sub_I_a.cpp () from ./a.so
17 # call_init (...) at dl-init.c:72
18 # _dl_init (...) at dl-init.c:119
19 # _dl_start_user () from ld-linux-x86-64.so.2

20 // clang++ -shared -fPIC  a.cpp -o a.so

21 _GLOBAL__sub_I_a.cpp:
22   mov    %gs:0x10,%r15

23 void print_in_a(){
24   cout << __FUNCTION__ << endl;
25 }
```

Fig. 12. Register clashing in initializing SHADESMAR-protected shared libraries.

In addition, we have also encountered program crashes in applying SHADESMAR to run four C++ benchmarks in SPEC CPU2006 (highlighted in **bold** in Table 3). Now, we analyze the program crashes for `447.dealII` and a SHADESMAR-protected shared library for a code example.

Figure 12 gives a simple example illustrating how a SHADESMAR-protected shared library can cause a program crash during its initialization in the dynamic linker (`ld-linux-x86-64.so.2` in line 1). On our computing system, `ld-linux-x86-64.so.2` is a soft link to `/lib/x86_64-linux-gnu/ld-2.27.so`, an unprotected system library released with Ubuntu 18.04.

The unprotected `_dl_init()` in lines 2-14 is a function in `ld-2.27.so` used to initialize the shared libraries for a program. By regarding `r15` as a callee-save register, this function saves `r15` in line 3 and restores it in line 12. In line 4, its first argument, `main_map` saved in register `rdi`, is moved into `r15`. The loop in lines 6-11 will invoke `call_init()` to initialize all the shared libraries one by one. As shown in line 10, `l_initfini` is a field in the structure `link_map`, with its offset (`0x3b8`) given in line 7. In lines 16-19, we give the stack trace when a SHADESMAR-protected shared

library, named a.so, is initialized. For simplicity, we have included only a simple function (lines 23-25) in its source file (a.cpp). The function _GLOBAL__sub_I_a.cpp() in line 21 is generated by clang++. The move instruction in line 22 is added by SHADESMAR to save its shadow stack pointer into r15. When this function returns, we are still in the loop in line 11. In the next iteration, _dl_init() tries to initialize the next shared library. Now, r15 in line 7 points to the shadow stack, rather than the structure link_map expected by _dl_init(). This is a register clash. Subsequently, a null-pointer dereference (program crash) in line 9 is observed, where both rdx and rax contain 0 in this example.

Let us analyze the program crash of the SPEC CPU2006 C++ program, 447.dealII, protected by SHADESMAR (Table 3). This happens when the program is about to exit. The unprotected function __run_exit_handlers() (in libc-2.27.so) is called, which also uses r15 to access an object. Then, a SHADESMAR-protected C++ destructor ~Subscriptor() is invoked, where r15 is assumed to point to the shadow stack, thus leading to a register clash. Specifically, the two move instructions at its prologue (as lines 11-12 in Figure 1) will corrupt the object pointed by r15 in the caller function __run_exit_handlers(). When the C++ destructor returns, a program crash ensues in __run_exit_handlers(), yielding the stack trace:

```
// The stack trace of 447.dealII
#  Subscriptor::~Subscriptor()
#  __run_exit_handlers()
#  __GI_exit()
#  __libc_start_main()
#  _start()
```

The program crashes for the other three SHADESMAR-protected SPEC CPU2006 C++ programs (Table 3), 453.povray, 471.omnetpp, and 483.xalancbmk, can be analyzed similarly. They happen after some SHADESMAR-protected C++ destructors for global objects are called in the unprotected library function __run_exit_handlers().

We have also observed that the 18 C/C++ SPEC CPU2006 benchmarks (excluding 447.dealII) in the SHADESMAR paper [12] are evaluated on Ubuntu 16.04. The version of the system libraries on their platform may be different from ours (ld-2.27.so and libc-2.27.so). That may explain why they did not encounter any program crash in their evaluation. *This again demonstrates that compatibility is a major issue when deploying a REG-TLS-based hardening tool on one machine even after it has been fully tested on another machine.*

Therefore, a REG-TLS-based approach such as SHADESMAR [12] suffers from poor compatibility on general-purpose ×86-64 platforms due to closed-source libraries, assembly files and unprotected code, but may be more feasible on customizable embedded systems where a register can be dedicated to serve as the shadow stack pointer [3]. In contrast, our STK-TLS-based BUSTK can provide compatibility, as evaluated in Table 6 for a range of fundamental projects, including several large multi-threaded applications like Firefox.

5.3 Security

BUSTK provides meaningful security by re-randomizing the return addresses on the shadow stack continuously and efficiently to obsolete a previously leaked value of $\mathscr{D}$, making it extremely unlikely for the attacker to hijack return addresses in server programs (Section 3.3) . However, the security protection provided by BUSTK for browsers is relatively weak, as acknowledged in Section 1.

On ×86, a return address pushed by a call instruction may be modified by the attacker before it is fetched in the prologue of the protected function (called). This can lead to a so-called **TOCTTOU (time-of-check to time-of-use)** problem. However, this problem does not exist on ARM, as the return address is saved in a register (rather than on the call stack) by hardware. One known exploit of this TOCTTOU time window on ×86 is reported in [13] in a locally-running environment. However, according to [12], it is non-trivial to exploit such a TOCTTOU window on ×86, in practice, as the attacker needs to rely on accurately timing the victim process. One possible mitigation is to instrument the program and insert an instruction to save the return address (or its abstract return ID) in a register before the call instruction [96]. However, this solution must sacrifice either compatibility or security in the presence of unprotected libraries. Nevertheless, as the network-access delay between a server and its clients is in the order of milliseconds [82], to the best of our knowledge, no such TOCTTOU attacks have been reported for server programs.

The effectiveness of a runtime re-randomization technique can be estimated in terms of its entropy provided. Both BARRA [96] and BUSTK focus on protecting return addresses (i.e., backward edges). In addition to the differences discussed in Section 1, BARRA has a default entropy of 20 bits while BUSTK uses 64 bits in protecting server programs. In contrast, SHUFFLER [82] can protect both backward and forward edges with 27-bit entropy by shuffling all code sections. Thus, SHUFFLER can provide better security for client-side programs than BUSTK, but at the expense of higher performance overheads. As for server programs where some physical network delays (tens of milliseconds [82]) exist, BUSTK can have larger entropy than SHUFFLER in protecting return addresses.

5.3.1 Protection for Server Programs. For an `Nginx`-like web server, **RTT (Round Trip Time)**, which is the time from when a data packet is sent to when its response is received [81], is a performance-critical metric. By using `ping` (based on the ICMP protocol) to benchmark `www.google.com`, the average RTT is found to be about 1.767 ms. By using also the `ApacheBench` tool to benchmark `www.google.com`, an HTTP request is found to take about 140 ms to complete. These are consistent with what is reported in previous research [82] that server programs often sit tens of milliseconds away from an attacker machine. Therefore, our adaptive 1-ms re-randomization policy is fast enough in mitigating ROP attacks against server programs by making $\mathscr{D}$ leaked earlier obsolete before it is used. Below we elaborate on the security protection provided for server programs:

- **Blind ROP.** Blind ROP [6] represents a clone-probing attack, which can infer the layout of an `Nginx` server process by repeatedly probing its worker processes, which are forked and re-spawned with the same layout of the parent process. By exploiting a stack buffer overflow, the attacker can corrupt a return address with a malicious input under his control. When the attacked function returns, whether the worker process crashes or not (after probing) can be used as a timing channel to infer and accumulate the memory layout information about the parent process repeatedly, finally bypassing the information hiding protection due to ASLR. For a BUSTK-protected `Nginx` server, its return addresses are continuously re-randomized by adding a random value $\mathscr{D}$ and stored in our shadow stacks. These shadow stacks are effective in protecting return addresses from stack buffer overflows (when the attacked function returns, the saved return address in the shadow stack is used, rather than the corrupted one in the call stack). In addition, Blind ROP cannot infer $\mathscr{D}$ through the side channel regarding whether the server has crashed or not. As a result, the return addresses of a BUSTK-protected `Nginx` server are protected by BUSTK from such clone-probing attacks.
- **Thread Spraying.** Thread spraying attacks [32, 59] are used to expose SAFESTACK [43] hidden in a large virtual memory space by creating (spraying) lots of threads. With the large

virtual memory space consumed this way, the entropy of information hiding is also reduced gradually, such that the attacker can locate the position of SafeStack and then corrupt the return addresses saved on SafeStack. As for a Bustk-protected server, we rely on fast re-randomization of $\mathscr{D}$ to make the leaked $\mathscr{D}$ obsolete by leveraging the fact that the network-access delays are in the order of tens of milliseconds, instead of the information hiding provided by the large virtual memory space. In our design, just locating the buddy stack is thus insufficient to hijack the return addresses protected by Bustk. For this reason, the shadow stack is not hidden at all but exposed (at a known constant offset to its call stack).

- **Allocation Oracles.** By exploiting memory corruption vulnerabilities to overwrite the arguments to memory allocation functions (e.g., `malloc()`), allocation oracles [32, 59] can hoax the attacked server to repeatedly allocate large chunks of memory. Then, the attacker can use the response status from the server (HTTP 200 or 500), fault side channels (whether the server has crashed or not), and timing side channels (cache hit or miss ratios) to infer the memory layout of the attacked server. In this way, the attacker can also expose hidden safe regions like SafeStack. Similarly, for a Bustk-protected server, the network-access delays between a server and its clients will still be an obstacle for the attacker to divert the backward edges of a Bustk-protected server due to our adaptive 1-ms re-randomization policy used.
- **Side-Channel Attacks.** By exploiting hardware (e.g., cache behavior [45, 86]) or software vulnerabilities [9, 10], side-channel attacks can gather information about the secret values used in a security-critical system. The attacker can undoubtedly launch side-channel attacks to obtain, for example, $\mathscr{D}$ stored in the buddy stack. However, for a Bustk-protected server, Bustk's fast re-randomization of $\mathscr{D}$ can make obsolete the leaked $\mathscr{D}$ (as discussed above).

5.3.2 Protection for Client-Side Programs. Locally running client-side programs can be divided into two classes: the programs running under a dynamic script environment (e.g., JavaScript in browsers and ActionScript in Flash players), and the programs running without a script environment (e.g., most of the GNU core utilities in Linux). In the former case, Bustk can also be helpful in protecting return addresses by mitigating ROP attacks that exploit traditional stack buffer overflows reported in browsers [53, 54]. However, a 1-ms re-randomization period will no longer be adequate to make the leaked $\mathscr{D}$ obsolete. For example, under a disclosure-aided exploit [29, 68], the attacker can quickly read $\mathscr{D}$ on the buddy stack and then overwrite a shadowed return address with a malicious return address (randomized using $\mathscr{D}$). In this case, the reported network-access delays between a server and a client (tens of milliseconds) do not exist for a web browser. For the programs running without a script environment, however, Bustk can still provide strong protection. As Bustk can perform runtime re-randomization triggered at a hooked I/O function, the attacker will still find it difficult to use maliciously-formatted inputs to hijack the Bustk-protected return addresses, as there is not any dynamically-running script for the attacker to peek $\mathscr{D}$ at runtime.

5.4 Shadow Stack Memory Overhead

We show that the memory overhead incurred by a parallel shadow stack is quite small (against the conventional wisdom), costing tens of KBs on average, in real-world single- and multi-threaded applications. As a result, we recommend the community to re-think about this shadow stack mechanism, especially since it can be adopted successfully in Bustk to offer compatibility (Section 5.2).

Table 8 gives the call stack usage information collected for four groups of applications, Tools, IDEs, Browsers, and Servers, by using a tool developed based on Intel Pin [49], as described in Section 4.5. For each program, the number of threads (or processes if it is single-threaded) executed is listed. For a program execution, we are interested in its two types of stack usage information: (1) call stack size (the physical memory consumed in bytes), and (2) call stack depth (the maximum

Table 8. The Call Stack Usage Information for Four Groups of Applications

Group	Application	#Threads/#Processes	Call Stack Sizes (Bytes)			Call Stack Depths		
			MIN	MAX	AVG	MIN	MAX	AVG
Tools	clang-7	1,665	19,344	163,424	60,326	28	358	88
	clang	553	19,840	19,888	19,864	23	23	23
	clang++	1,131	19,840	19,888	19,864	23	23	23
	/usr/bin/find	19	5,696	5,696	5,696	15	15	15
	/usr/bin/ld	19	13,728	13,728	13,728	21	24	22
	/usr/bin/xargs	19	3,336	3,336	3,336	15	15	15
	/bin/hostname	1	3,336	3,336	3,336	13	13	13
	/bin/rm	57	3,336	3,336	3,336	12	12	12
	/bin/sh	432	3,336	5,984	3,452	14	14	14
	runspec	1	270,640	270,640	270,640	222	222	222
	specmake	57	17,968	53,040	21,627	82	82	82
IDEs	Eclipse	3	1,872	44,304	16,592	9	233	88
	jdk1.8/bin/java	86	1,216	958,864	31,292	9	8,702	177
	Python3.6	1	80,560	80,560	80,560	183	183	183
Browsers	Firefox79.0	121	560	133,344	13,546	7	277	37
	Chrome84.0	41	1,056	112,640	15,033	9	276	40
Servers	Nginx1.18	5	9,488	9,488	9,488	24	43	39
	Apache Httpd2.4.46	28	4,416	35,376	32,251	12	30	28

number of frames on the stack). By executing a program multiple times (against possibly different inputs), the minimum, maximum, and average for each metric are obtained.

Let us examine these applications briefly. In the first group, we see the tools used for building the SPEC CPU2006 benchmarks. Take `clang-7` as an example. During its 1,665 executions, the minimum, maximum and average call stack sizes are 19,344, 163,424, and 60,326 bytes, respectively. The second group contains the three applications involved when `Eclipse` is used to open a Java project, `Pysonar2` [78]. During the 86 executions of JVM, its call stack sizes range from 1,216 bytes to 958,864 bytes with an average of only 31,292 bytes. Surprisingly, the two popular browsers, `Firefox` and `Chrome`, in the third group, rely on much smaller average call stack sizes, 13,546 bytes and 15,033 bytes, respectively, when visiting [23] and [57]. In the last group, the `Nginx` server uses a call stack of 9,488 bytes only in handling the requests from `ApacheBench`, while the `Apache Httpd` server needs to use a slightly larger call stack (32,251 bytes, on average).

The average call stack size in these applications is tens of KBs. Thus, the average memory overhead per shadow stack is also tens of KBs. This appears to be an acceptable overhead on computer systems with 8GB or higher physical memory nowadays when defending against control-flow hijacking attacks targeting return addresses. Even if we do allocate a large virtual space of 32MB for a buddy stack per thread as shown in Figure 6, BUSTK will still be feasible on ×86-64, where a process now can own a 47-bit user space (128 TB). On modern operating systems with demand paging, only two physical pages in the two metadata sections (allocated in a 16MB virtual memory region) are actually accessed in our current implementation. Given that a user-space process on ×86-64 has a 47-bit virtual space (128 TB) and a process rarely has more than 1,024 threads [32], a 32MB-size buddy stack does not really limit the number of threads needed, in practice.

5.5 Code Size Overhead

Table 9 presents the code size increases of the SHADESMAR- and BUSTK-protected `Nginx` and `Apache` web servers. For `Nginx`, both mitigation tools have small code size overheads (1.33% for SHADESMAR and 0.58% for BUSTK). As for `Apache`, we give the total code size increase caused by each tool for

Table 9. Code Size Increases of Shadesmar and Bustk in Web Servers

	Vanilla (Bytes)	Shadesmar	Bustk
Nginx	3,888,072	1.33%	0.58%
Apache Httpd + its 91 shared libraries	3,634,304	4.13%	5.64%

Table 10. Code Size Increases of Shadesmar and Bustk in SPEC CPU2006

Benchmark	Vanilla (Bytes)	Shadesmar	Bustk
400.perlbench	1,304,720	13.01%	7.85%
401.bzip2	102,256	17.71%	4.01%
403.gcc	3,976,184	8.18%	7.00%
429.mcf	22,824	43.67%	17.95%
433.milc	172,608	17.64%	7.12%
444.namd	337,112	6.62%	1.22%
445.gobmk	4,040,872	2.07%	3.75%
447.dealII	4,550,464	25.24%	8.19%
450.soplex	502,664	20.74%	9.78%
453.povray	1,330,832	13.37%	7.08%
456.hmmer	372,128	12.57%	7.70%
458.sjeng	163,504	18.65%	7.52%
462.libquantum	59,440	30.44%	13.78%
464.h264ref	836,200	9.55%	3.92%
470.lbm	21,992	45.51%	0.00%
471.omnetpp	821,408	23.65%	12.97%
473.astar	60,112	30.32%	6.72%
482.sphinx3	234,256	14.64%	8.74%
483.xalancbmk	6,125,272	33.60%	11.50%
Average		20.38%	7.73%

the main executable and its 91 shared libraries together. Bustk incurs a slightly larger code size overhead than Shadesmar (5.64% vs. 4.13%). The ratio of the code size overheads from these two tools may fluctuate slightly from application to application, as Shadesmar chooses to protect only the functions that may write into memories (line 269 in `×86ShadowStackReg.cpp` [66]). However, such an optimization may trade security for performance and code size.

Table 10 shows that Bustk has smaller code size increases than Shadesmar for all the 19 SPEC CPU2006 benchmarks except for `445.gobmk`. Note that Shadesmar will generate larger code sizes if its MPX-/MPK-protection is turned on. We have seen an average code size increase of 20.38% for Shadesmar and of 7.73% for Bustk. One interesting observation is that `470.lbm`, once protected by Bustk, has exactly the same size as before (`21,992` bytes). Their binary files do have different text sections, but just happen to have the same code size due to section alignment. By protecting return addresses with compact shadow stacks, Shadesmar adds some auxiliary functions (e.g., `init_shadow_stack_reg()`) in an instrumented executable for initializing its shadow stack register. For some small executables, such as 429.mcf, 462.libquantum, 470.lbm, and 473.astar, these additional functions have led to relatively large code size increases. In addition, Shadesmar also needs to add additional instructions for handling stack unwinding for C++ exceptions and setjmp/longjmp in C.

Now, let us discuss the code size increases of `Firefox` and its shared libraries caused by Bustk. Recall that Shadesmar cannot build `Firefox` due to register clashes. Table 11 gives the code size

Table 11. Code Size Increases of Bustk in `Firefox79.0`

	Vanilla (Bytes)	Bustk
Firefox	745,008	9.35%
gmp-clearkey/0.1/libclearkey.so	72,088	11.36%
gtk2/libmozgtk.so	18,328	0.00%
libfreeblpriv3.so	682,464	5.40%
liblgpllibs.so	43,128	9.50%
libmozavcodec.so	2,204,344	5.20%
libmozavutil.so	220,008	9.31%
libmozgtk.so	6,040	0.00%
libmozsandbox.so	154,392	7.96%
libmozsqlite3.so	1,330,152	5.54%
libmozwayland.so	10,056	40.73%
libnspr4.so	231,288	14.17%
libnss3.so	653,992	15.66%
libnssckbi.so	461,400	4.44%
libnssutil3.so	179,304	9.14%
libplc4.so	18,656	21.96%
libplds4.so	14,472	0.00%
libsmime3.so	155,848	15.77%
libsoftokn3.so	301,512	13.58%
libssl3.so	365,896	12.31%
libxul.so	124,408,768	10.71%
Average		10.58%

increases for `Firefox79.0` and its 20 shared libraries with respect to the code sizes obtained by vanilla LLVM 7.0. On average, each instrumented file is 10.58% larger than its vanilla one. Interestingly, `gtk2/libmozgtk.so`, `libmozgtk.so`, and `libplds4.so`, have exactly the same code sizes as their corresponding vanilla ones. Just as in the case of `470.lbm` discussed in Section 5.5, each of these three shared libraries differs in text sections from its vanilla one, but happens to have exactly the same code size due to section alignment.

We have also evaluated the performance overheads and code size increases of `Bustk.Global` (given in Figure 10) for the 19 SPEC CPU2006 benchmarks. The average performance overhead of `Bustk.Global` is 6.65% while its average code size increase is 19.19%. By comparing with Shadesmar and Bustk (Tables 3 and 10), `Bustk.Global` is comparable with Shadesmar (even though its expensive MPX-/MPK-based protection mechanism has been turned off as described in Section 5), but somewhat more costly than Bustk in terms of both additional performance and code overheads incurred. Therefore, we recommend Bustk to be adopted, despite its use of hard-coded stack sizes in hardened binaries (as discussed in Section 4.2).

5.6 A Syntax Error Found in LLVM-Generated Assembly

As we have implemented STK-TLS and STK-MSR as an assembly instrumentator (Figure 7), the assembly code generated by LLVM 7.0 will be finally assembled by the native assembler `AS` (/usr/bin/as on our machine). We find a syntax error in the assembly code generated from the C++ source file `gfx/skia/skia/third_party/skcms/skcms.cc` in `Firefox79.0`. As shown below, the instruction in line 2 is generated by LLVM 7.0 and can also be accepted by its own integrated

assembler. But the GNU assembler AS (/usr/bin/as) complains about its syntactic correctness. In line 4, deleting the white space in "{%k1} {z}" has made both happy.

```
1 # Generated and accepted by LLVM 7.0
2 vmovdqu16 %zmm5, %zmm5 {%k1} {z}
3 # Accepted by both LLVM 7.0 and GNU AS
4 vmovdqu16 %zmm5, %zmm5 {%k1}{z}
```

A similar issue is encountered if LLVM 10.0 is used for building `Firefox79.0`. As shown below, the `vdivps` instruction in line 6, which is generated by LLVM 10.0, is also flagged as a syntax error by the GNU assembler.

```
5 # Generated and accepted by LLVM 10.0
6 vdivps %zmm10, %zmm0, %zmm0 {%k1} {z}
7 # Accepted by both LLVM 10.0 and GNU AS
8 vdivps %zmm10, %zmm0, %zmm0 {%k1}{z}
```

By checking Intel 64 and IA-32 architectures software developer's manuals [36], we find that the white space (in lines 2 and 6) generated by LLVM is redundant.

This also shows one merit of our BUSTK design depicted in Figure 7: switching between different LLVM versions can be easily done in our compiler wrappers, with `bustk-rustc` illustrated in Figure 8. Currently, we have tested LLVM 7.0 and LLVM 10.0. Both can be used to build and run successfully a BUSTK-protected `Firefox79.0` browser.

5.7 Discussions

STK-TLS represents a novel stack-based TLS mechanism. We can also integrate it into a programming language by introducing a keyword, `__stk_tls`, to declare thread-local virtual registers.

STK-MSR can also be used to protect programs running on embedded systems (e.g., 32-bit Cortex M3/M4 processors used in IoT devices [20, 70, 92]), with compact shadow stacks, since embedded systems can be customized, making a REG-TLS-based protection scheme possible [3]. Due to constraints such as power and cost, embedded systems often lack advanced memory protections such as MMU [36], MPX [35], and MPK [63]. Without MMU, the widely-adopted information hiding mechanism within a large virtual memory space (in 64-bit desktop or server environments) is not available for these embedded systems. By design, BUSTK does not hide its shadow stacks as they sit right below the corresponding call stacks shadowed (Figure 6). As BUSTK is compatible with ASLR [75], security provided by information hiding (due to ASLR) is still available in BUSTK. Unlike the prior art such as SHADESMAR [12], BUSTK improves its security further by taking advantage of fast runtime re-randomization of $\mathscr{D}$ accomplished by STK-MSR. As STK-MSR itself does not need an MMU-based large virtual memory space, it can still be used to protect embedded systems without MMU. Note that STK-MSR uses `mprotect()` (Figure 11) to change the access protections of memory pages, which can be similarly supported by an MPU [70] (available in the popular 32-bit Cortex MCUs). For other embedded systems without MPUs, a hardware timer may be used to trigger runtime re-randomization given that a dedicated register can now be used for storing $\mathscr{D}$ in order to support STK-MSR (Figure 5). In addition, most of embedded systems are neither powerful enough nor designed to support desktop applications like browsers and Flash players. Therefore, we often do not need to consider the security threats of locally-running JavaScript or ActionScript programs on these systems. Therefore, STK-MSR can be particularly invaluable on these embedded systems.

For a runtime re-randomization technique, both the mechanism for implementing it and the policy for activating it are needed in providing integrity protection for the shadow stack. In the case

of Bustk, when and where to call `runtime_rerandomize()` in Figure 11 is a policy issue. But, first and foremost, we must have a highly efficient re-randomization mechanism available. In this paper, we focus mainly on developing a novel mechanism itself. Although we have provided an adaptive policy, `Bustk.n-ms`, wiser policies may be needed for protecting a given application. For example, cross-thread stack-smashing attacks [85] can be further mitigated through re-randomization by extending `Bustk.n-ms` to hook also some frequently executed functions (using `LD_PRELOAD`), adding calls to `runtime_rerandomize()` in strategically selected security-critical points in source code, and/or making the shadow stack of a non-running (i.e., waiting/blocked/sleeping) thread read-only for multi-threaded programs by the OS thread scheduler. These extensions are feasible as `runtime_rerandomize()` happens on the order of microseconds. We leave this policy issue to future work, as it is application-dependent (as also discussed in Section 4.3).

6 RELATED WORK

We review the past work that is directly relevant to Bustk, by focusing on call stack protection, **control flow integrity (CFI)**, and (re-)randomization in protecting return addresses.

Call Stack Protection. After Aleph One's seminal work on stack smashing [60], many defenses have been proposed to protect backward edges, i.e., return addresses on the call stack. StackShield [67] is one of the first shadow stack implementations to counteract stack smashing attacks. RAD [17] is a compile-time solution to thwart buffer overflow attacks. StackGuard [19] tests whether the canary inserted into a stack frame has been corrupted or not before the protected function returns. StackGuard can defend against sequential buffer overflows but is ineffective in the case of arbitrary writes. Traditional (parallel) shadow stacks [17, 21, 84] rely on information hiding and are thus vulnerable to information disclosure and side channel attacks. In addition, requiring the same fixed offset from the call stack to the shadow stack for all the threads is ill-suited for multi-threading. Switching to compact shadow stacks supports multi-threading well [12, 84], but using a dedicated register to maintain the separate shadow stack pointer sacrifices compatibility on ×86-64 platforms (as evaluated in this paper). Note that such a register-based scheme can certainly also be applicable to parallel shadow stacks [12]. As part of the **code pointer integrity (CPI)** project, SafeStack [32, 43] moves unsafe stack variables to a separate stack, but also at the expense of compatibility [12]. A leak-resilient dual stack scheme is proposed in [94], which can protect backward edges with a negligible overhead. But the use of $r15$ as a dedicated register on ×86 also sacrifices its compatibility. In contrast, Bustk is compatible with unprotected code as it is based on STK-TLS, a novel mechanism proposed for providing TLS in this paper.

Compared with Shadesmar [12], Bustk makes different tradeoffs among performance, compatibility, and security. Bustk keeps its instrumentation overheads low by bundling the call stack, its parallel shadow stack, and the related TLS metadata (in dual redundancy) together in one buddy stack design, avoiding the need to both maintain the shadow stack pointer and handle stack unwinding (for exception handling), as required in Shadesmar. This novel design also enables Bustk to achieve compatibility with not only multi-threading (as in Shadesmar) but also unprotected code (in contrast to the register clashes that may occur in Shadesmar). Unlike Shadesmar, Bustk relies on fast runtime re-randomization of $\mathscr{D}$ to provide strong protection for server programs, and usable but weaker protection for browsers. Finally, our work reveals that parallel shadow stacks do not lead to high memory overheads for many real-world applications running on ×86-64.

Intel has recently released a new processor, Intel Tiger Lake [80], to provide hardware support for the shadow stack mechanism. But market penetration is often expected to take 6+ years [13]. For programs running on embedded systems, their return addresses can be protected by μRAI [3].

As embedded systems are often fully customizable (with some dedicated registers available), it becomes more feasible to maintain compact shadow stacks on these systems. We plan to further improve the performance of our buddy stack design and apply our runtime re-randomization technique to counteract information disclosure attacks on these platforms [93].

CFI. The seminal work on CFI was proposed by Abadi et al. in 2005 [1] for protecting forward edges, i.e., function pointers and virtual table pointers, spurring a great deal of subsequent research. Coarse-grained CFI schemes [16, 62, 90] require low-overhead runtime checks, but are bypassable [22, 31]. Alternatively, fine-grained CFI mechanisms [28, 40, 58, 76, 89] are explored, utilizing the high-level semantics of a protected program to monitor its indirect controls more precisely but often more expensively [11, 40, 46].

Despite many advances made on CFI, dynamic linking remains a major challenge for CFI compatibility [85] and incremental deployability remains to be an Achilles heel of CFI [64], since control flow graph edges that span different modules (e.g., shared libraries) may not be available statically.

(Re-)Randomization. ASLR [75], a coarse-grained randomization mechanism, performs a single, per-process randomization that is applied before or at the process' load-time, targeting only the base addresses of code and data sections. To raise the bar further, fine-grained randomization schemes at the levels of functions, basic blocks, and instructions have also been studied [30, 39, 41, 48, 61]. Runtime code re-randomization [5, 82] can be used to mitigate JIT-ROP or clone-probing attacks [6, 47, 68]. However, pointer tracking required is time consuming and troublesome [47]. BARRA [96] combines shadow stack, CFI, and runtime re-randomization to protect return addresses for single-threaded programs, by trading security for compatibility due to whole-program analysis used.

Code sharing and fine-grained software diversity have been regarded as being contradictory to each other. Backes et al. [4] consider it practical to combine both together. However, Koo et al. [42] have recently observed that fine-grained randomization schemes are still academia prototypes, without the widespread adoption due to their incompatibility with existing software distribution models. One solution is to generate transformation-assisting metadata at the software-vendor side and reorder basic blocks at the end-user side [42].

In BUSTK, its runtime re-randomization mechanism is applied to the data sections (shadow stacks) of the program, with the code sections of a BUSTK-protected program being not diversified. All the BUSTK-protected shared libraries can still naturally be mapped into different processes and shared among them as usual. In other words, the software diversity provided by BUSTK is through data plane rather than code plane. Therefore, BUSTK will not affect code sharing at all.

7 CONCLUSION

In this paper, we have introduced a new stack-based **TLS (thread-local storage)** mechanism (STK-TLS), which can be used in scenarios where frequent accesses to thread-local storage need to be done efficiently and compatibly. In addition, we have also introduced a new microsecond-level runtime re-randomization mechanism (STK-MSR) for mitigating information disclosure attacks against shadow stacks. By combining STK-TLS and STK-MSR, BUSTK represents a new approach for mitigating ROP attacks, that is compatible with existing code, highly performant, and provides meaningful security for single- and multi-threaded server programs.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 340–353.

[2] Adobe. 2021. Adobe Flash Player. https://get.adobe.com/flashplayer/.

[3] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. 2020. μRAI: Securing embedded systems with return address integrity. In *Network and Distributed System Security Symposium*. Internet Society, Reston, Virginia, USA, 1–18.

[4] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX, Berkeley, California, USA, 433–447.

[5] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 268–279.

[6] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 227–242.

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

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 2329–2344.

[9] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT leaks: Inducing timing side channels through just-in-time compilation. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 96–111.

[10] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM fuzzing for JIT-induced side-channel detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, New York, 1011–1023.

[11] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CFIXX: Object type integrity for C++ virtual dispatch. In *Network and Distributed System Security Symposium*. Internet Society, Reston, Virginia, USA, 1–14.

[12] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 985–999.

[13] Microsoft Security Response Center. 2018. The Evolution of CFI Attacks and Defenses. https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2018_02_OffensiveCon.

[14] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, New York, 559–572.

[15] Karl Chen and David Wagner. 2007. Large-scale analysis of format string vulnerabilities in Debian Linux. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*. ACM, New York, 75–84.

[16] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H. Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attack. In *Network and Distributed System Security Symposium*. Internet Society, Reston, Virginia, USA, 1–14.

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

[18] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 952–963.

[19] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*. USENIX, Berkeley, California, USA, 1–16.

[20] Hong-Ning Dai, Zibin Zheng, and Yan Zhang. 2019. Blockchain for Internet of Things: A survey. *IEEE Internet of Things Journal* 6, 5 (2019), 8076–8094.

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

[22] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX, Berkeley, California, USA, 401–416.

[23] Bram Debouvere. 2021. Bmark. https://www.wirple.com/bmark/.

[24] Google Developers. 2021. Octane2.0 Javascript Benchmark. https://chromium.github.io/octane/.

[25] JetStream2 Developers. 2021. JetStream2: JavaScript and Web Assembly benchmarks. https://browserbench.org/JetStream/.

[26] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery.. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 151–163.

[27] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. 2015. Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 781–796.

[28] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, New York, 329–340.

[29] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In *Network and Distributed System Security Symposium*. Internet Society, Reston, Virginia, USA, 1–15.

[30] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Security Symposium*. USENIX, Berkeley, California, USA, 40–55.

[31] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX, Berkeley, California, USA, 417–432.

[32] Enes Goktas, Angelos Oikonomopoulos, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Bypassing clang's SafeStack for fun and profit. In *Black Hat Europe*. Black Hat Conference, London, 1–76.

[33] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 1470–1486.

[34] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 969–986.

[35] Intel. 2013. Introduction to Intel Memory Protection Extensions. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html.

[36] Intel. 2021. Intel 64 and IA-32 Architectures Software Developer's Manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[37] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 1868–1882.

[38] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient detection of type confusion errors for C++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 2373–2387.

[39] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*. ACM, New York, 272–280.

[40] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive control flow integrity. In *Proceedings of the 28th USENIX Security Symposium*. USENIX, Berkeley, California, USA, 195–211.

[41] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*. Applied Computer Security Associates, Silver Spring, MD, USA, 339–348.

[42] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 461–477.

[43] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX, Berkeley, California, USA, 147–163.

[44] Jinfeng Li, Liwei Chen, Qizhen Xu, Linan Tian, Gang Shi, Kai Chen, and Dan Meng. 2020. Zipper stack: Shadow stacks without shadow. In *European Symposium on Research in Computer Security*. Springer, New York, 338–358.

[45] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 605–622.

[46] Kangjie Lu and Hong Hu. 2019. Where does it go? Refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 1867–1881.

[47] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to make ASLR win the clone wars: Runtime re-randomization. In *Network and Distributed System Security Symposium*. Internet Society, Reston, Virginia, USA, 1–15.

[48] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 280–291.

[49] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 190–200.

[50] Matz Michael, Hubicka Jan, Jaeger Andreas, and Mark Mitchell. 2021. System V Application Binary Interface. https://uclibc.org/docs/psABI-$\times$86_64.pdf.

[51] Microsoft. 2018. Data Execution Prevention. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[52] Mozilla. 2017. Thoughts on Mozilla using Closed-Source Software. https://discourse.mozilla.org/t/thoughts-on-mozilla-using-closed-source-software/19000.

[53] Mozilla. 2021. Security vulnerabilities fixed in Firefox 70. https://www.mozilla.org/en-US/security/advisories/mfsa2019-34.

[54] Mozilla. 2021. Stack-Based-Buffer Overflow when parsing specific URLs in Firefox 68.0. https://bugzilla.mozilla.org/show_bug.cgi?id=1565912.

[55] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 245–258.

[56] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: Compiler enforced temporal safety for C. In *ACM Sigplan Notices*. ACM, New York, 31–40.

[57] NASA. 2021. Earth From Space LIVE Feed. https://www.youtube.com/watch?v=EEIk7gwjgIM&ab_channel=SpaceVideos.

[58] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 914–926.

[59] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking holes in information hiding. In *Proceedings of the 25th USENIX Security Symposium*. USENIX, Berkeley, California, USA, 121–138.

[60] Aleph One. 1996. Smashing The Stack For Fun And Profit. http://phrack.org/issues/49/14.html.

[61] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 601–615.

[62] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*. USENIX, Berkeley, California, USA, 447–462.

[63] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software abstraction for Intel memory protection keys. In *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX, Berkeley, California, USA, 241–254.

[64] Aravind Prakash and Heng Yin. 2015. Defeating ROP through denial of stack pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference*. Applied Computer Security Associates, Silver Spring, MD, USA, 111–120.

[65] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the ×86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, New York, 552–561.

[66] Shadesmar. 2019. ×86ShadowStackReg.cpp. https://github.com/HexHive/ShadowStack/blob/master/Compiler-Impl/src/$\times$86ShadowStackReg.cpp.

[67] Stack Shield. 2000. A stack smashing technique protection tool for Linux. https://www.angelfire.com/sk/stackshield/info.html.

[68] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 574–588.

[69] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*.

[70] STMicroelectronics. 2021. STM32 32-bit Arm Cortex MCUs. https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html.
[71] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, New York, 160–170.
[72] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, New York, 460–473.
[73] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, New York, 265–266.
[74] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 48–62.
[75] The PaX Team. 2001. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt.
[76] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX, Berkeley, California, USA, 941–955.
[77] Drepper Ulrich. 2013. ELF Handling for Thread-Local Storage. https://akkadia.org/drepper/tls.pdf.
[78] Yin Wang. 2019. PySonar2: An advanced semantic indexer for Python. https://github.com/yinwang0/pysonar2.
[79] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy ×86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, New York, 157–168.
[80] Wikipedia. 2020. Tiger Lake. https://en.wikipedia.org/wiki/Tiger_Lake_(microprocessor).
[81] Wikipedia. 2021. Round Trip Time. https://en.wikipedia.org/wiki/Round-trip_delay.
[82] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Berkeley, California, USA, 367–382.
[83] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 133–147.
[84] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. 2002. Architecture support for defending against buffer overflow attacks. *Coordinated Science Laboratory Report no. UILU-ENG-02-2205, CRHC-02-05* (2002), 1–8.
[85] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. 2019. CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *Proceedings of the 28th USENIX Security Symposium*. USENIX, Berkeley, California, USA, 1805–1821.
[86] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX, Berkeley, California, USA, 719–732.
[87] Michal Zalewski. 2021. American fuzzy lop (AFL) fuzzer. http://lcamtuf.coredump.cx/afl/.
[88] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting virtual function tables' integrity. In *Network and Distributed System Security Symposium*. Internet Society, Reston, Virginia, USA, 1–15.
[89] Chao Zhang, Dawn Song, Scott A. Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining trust on virtual calls. In *Network and Distributed System Security Symposium*. Internet Society, Reston, Virginia, USA, 1–15.
[90] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington D.C., USA, 559–573.
[91] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. 2019. Pex: A permission check analysis framework for Linux kernel. In *Proceedings of the 28th Conference on USENIX Security Symposium*. USENIX, Berkeley, California, USA, 1205–1220.
[92] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* 105 (2020), 475–491.
[93] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. 2020. Silhouette: Efficient protected shadow stacks for embedded systems. In *Proceedings of the 29th USENIX Security Symposium*. USENIX, Berkeley, California, USA, 1219–1236.

[94] Philipp Zieris and Julian Horsch. 2018. A leak-resilient dual stack scheme for backward-edge control-flow integrity. In *Proceedings of the 2018 Asia Conference on Computer and Communications Security*. ACM, New York, 369–380.

[95] Changwei Zou, Yulei Sui, Hua Yan, and Jingling Xue. 2019. TCD: Statically detecting type confusion errors in C++ programs. In *Proceedings of the 30th International Symposium on Software Reliability Engineering*. IEEE, New York, 292–302.

[96] Changwei Zou and Jingling Xue. 2020. 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*. ACM, New York, 258–270.

Received March 2021; revised September 2021; accepted October 2021