Burn After Reading: A Shadow Stack with Microsecond-level Runtime Rerandomization for Protecting Return Addresses [*]

Changwei Zou
School of Computer Science and Engineering
UNSW Sydney, Australia

Jingling Xue
School of Computer Science and Engineering
UNSW Sydney, Australia

ABSTRACT

Return-oriented programming (ROP) is an effective code-reuse attack in which short code sequences (i.e., gadgets) ending in a ret instruction are found within existing binaries and then executed by taking control of the call stack. The shadow stack, control flow integrity (CFI) and code (re)randomization are three popular techniques for protecting programs against return address overwrites. However, existing runtime rerandomization techniques operate on concrete return addresses, requiring expensive pointer tracking.

By adding one level of indirection, we introduce BarRA, the first shadow stack mechanism that applies continuous runtime rerandomization to abstract return addresses for protecting their corresponding concrete return addresses (protected also by CFI), thus avoiding expensive pointer tracking. As a nice side-effect, BarRA naturally combines the shadow stack, CFI and runtime rerandomization in the same framework. The key novelty of BarRA, however, is that once some abstract return addresses are leaked, BarRA will enforce the burn-after-reading property by rerandomizing the mapping from the abstract to the concrete return address space in the order of microseconds instead of seconds required for rerandomizing a concrete return address space. As a result, BarRA can be used as a superior replacement for the shadow stack, as demonstrated by comparing both using the 19 C/C++ benchmarks in SPEC CPU2006 (totalling 2,047,447 LOC) and analyzing a proof-of-concept attack, provided that we can tolerate some slight binary code size increases (by an average of 29.44%) and are willing to use 8MB of dedicated memory for holding up to 2^{20} return addresses (on a 64-bit platform). Under an information leakage attack (for some return addresses), the shadow stack is always vulnerable but BarRA is significantly more resilient (by reducing an attacker's success rate to $\frac{1}{2^{20}}$ on average). In terms of the average performance overhead introduced, both are comparable: 6.09% (BarRA) vs. 5.38% (the shadow stack).

KEYWORDS

Shadow Stack, Runtime Rerandomization, Control Flow Integrity, Return-Oriented Programming

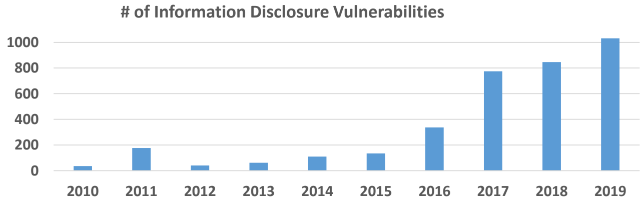


Figure 1: Rise of information disclosure vulnerabilities [13].

1 INTRODUCTION

Software security is becoming increasingly important due to increased reliance on computer systems. For performance reasons, C and C++ are still the de facto languages for implementing OS kernels, browsers and web servers. Due to their lack of memory safety, security vulnerabilities such as buffer overflows are frequently found in C/C++ software applications ranging from servers to embedded systems [31, 32, 45, 46, 51]. This has allowed an attacker to launch control-flow hijacking attacks to redirect execution to malicious code by modifying code pointers such as return addresses on the call stack [38]. Due to the widespread adoption of data execution prevention (DEP) techniques such as W⊕ X [29], code-injection attacks (which rely on malicious code injected into a program) [23] are no longer threatening. To circumvent DEP, code reuse attacks (which rely on malicious code formed from existing code fragments known as *gadgets*) are becoming more prevalent [5, 8, 42]. In particular, return-oriented programming (ROP) [42] is an effective code-reuse attack in which a return address on the call stack is modified to redirect execution to a sequence of gadgets with each ending in a ret instruction (*ROP gadgets*) to perform arbitrary malicious computations. It is thus imperative to develop mitigation techniques to protect programs against return address overwrites.

Problem Statement. A shadow stack [1, 3, 21] protects a function's return address on the call stack by hiding it in the shadow stack, which cannot be as easily modified by a buffer overflow happening on the call stack. However, this mechanism relies on information hiding (for the shadow stack), and consequently, is vulnerable to information disclosure and side-channel attacks [9, 16, 18, 36]. ASLR (Address Space Layout Randomization) [47], which randomizes the locations of modules at load time, is also vulnerable to information leakage attacks [9, 16, 18, 36]. As shown in Figure 1, the number of information disclosure vulnerabilities reported on the CVE website [13] has surged in recent years.

Control flow integrity (CFI) [1] can also be used to protect return addresses, by limiting all ROP gadgets to a set of predefined return addresses, but the resulting attack surface is still too large [7, 15].

[*] Thanks to all the reviewers for their valuable comments. This research is supported by an Australian Research Council grant (DP180104069).

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380439

Given that some information leaks are detected during program execution, we investigate how to apply continuous runtime rerandomization to make the shadow stack significantly more secure.

Challenges. There are three challenges faced. First, how do we provide lightweight runtime rerandomization, especially if it needs to be frequently performed during program execution? Second, how do we minimize the instrumentation overhead thus introduced for maintaining the shadow stack? Finally, how can we make the shadow stack significantly more secure by applying runtime rerandomization, and possibly, CFI at the same time?

Prior Work: Rerandomizing Concrete Return Addresses. Existing runtime rerandomization techniques [4, 26] operate on concrete return addresses, and consequently, are applied to the call stack directly. Once a return address on the call stack has been leaked, indicating that it may soon be replaced by the address to a ROP gadget, runtime rerandomization [4, 26] can be applied so that the address of the ROP gadget becomes invalidated, thereby mitigating the impact of information leakage. However, rerandomizing concrete return addresses requires expensive and difficult pointer tracking. For example, RuntimeASLR [26] (a state-of-the-art runtime rerandomizer) takes 35 seconds to track the pointers for the *nginx* web server. Due to its excessive performance overhead, RuntimeASLR rerandomizes a freshly-forked child process only once, at the time of *fork()*, by reusing the pointer tracking results of its parent (as the child process inherits the state of its parent just before it starts its execution). By failing to rerandomize a child process that has executed for a while, RuntimeASLR avoids the costs incurred by new time-consuming pointer tracking operations, but at the risk of being vulnerable to code-reuse attacks (Figure 1).

To the best of our knowledge, CFI [1] has been applied to protect forward edges, i.e., indirect calls via function pointer and virtual calls (instead of backward edges, i.e., return addresses). The research on forward-edge CFI [21, 48**?**] assumes usually that the shadow stack mechanism is used for enforcing backward-edge CFI.

This Work: Rerandomizing Abstract Return Addresses. By adding one level of indirection, we introduce the first shadow stack mechanism, BarRA, that applies a novel runtime rerandomization technique to rerandomize abstract return addresses in the shadow stack to protect their corresponding concrete return addresses, thereby avoiding expensive pointer tracking as required in RuntimeASLR [26]. Under some information leaks, BarRA will immediately rerandomize the mapping from the abstract to the concrete return address space in the order of microseconds instead of seconds as required by RuntimeASLR [26]. This enforces the *burn-after-reading* property, which requires all leaked (return address) information to be made obsolete via rerandomization. As a result, BarRA has made the traditional shadow stack significantly more secure while incurring a comparable instrumentation overhead on average. Finally, BarRA represents the first approach that protects programs against return address overwrites by combining the shadow stack, CFI and runtime rerandomization altogether.

This paper makes the following two major contributions:

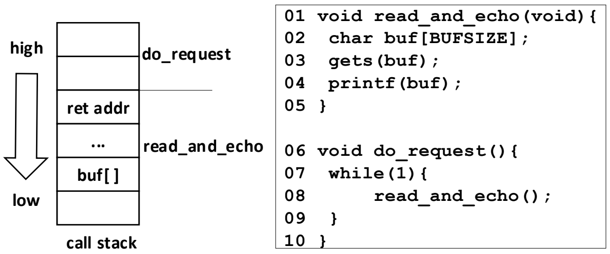
- We introduce a novel shadow stack mechanism that is capable of applying continuous microsecond-level runtime rerandomization for protecting return addresses, making the traditional shadow stack significantly more secure at comparable performance overheads (Sections 2 and 3).
- We have implemented BarRA as a software hardening tool and experimentally confirmed BarRA as a superior replacement for the traditional shadow stack (Section 4). In our evaluation, we have used all the 19 C/C++ benchmarks in SPEC CPU2006 (totalling 2,047,447 LOC) and a proof-of-concept attack. In the case of information leakage, the shadow stack is always vulnerable. However, if we can tolerate slight binary code size increases (by an average of 29.44%) and are willing to use 8MB of dedicated memory for holding up to 2^{20} return addresses (on a 64-bit platform), we can make BarRA significantly more resilient than the traditional shadow stack by reducing an attacker's success rate to $\frac{1}{2^{20}}$ on average. Both have comparable average performance overheads: 6.09% (BarRA) vs. 5.38% (the shadow stack).

2 BARRA: METHODOLOGY

We motivate our BarRA methodology by describing how we can transform a traditional shadow stack into a significantly more secure "burn-after-reading" shadow stack. Section 2.1 uses an example to explain how a buffer overflow can lead to return address overwrites on the call stack. Section 2.2 describes how this vulnerability in the example can be exploited to launch a ROP attack. Continuing with the same example, we describe why the traditional shadow stack mechanism is vulnerable to ROP attacks in the presence of information leakage (Section 2.3) and how our "burn-after-reading" shadow stack is significantly more secure (Section 2.4).

2.1 Buffer Overflow Vulnerabilities

Figure 2 illustrates how a buffer overflow bug can cause the return address of a function on the call stack to be modified.



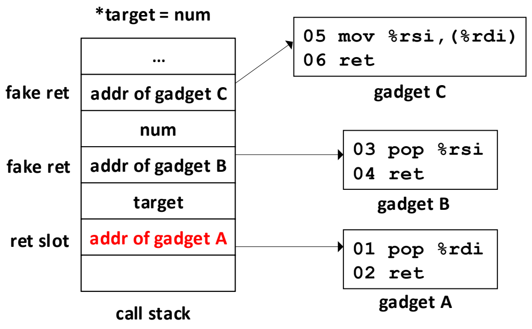
```
01 void read_and_echo(void){
02  char buf[BUFSIZE];
03  gets(buf);
04  printf(buf);
05 }

06 void do_request(){
07  while(1){
08      read_and_echo();
09  }
10 }
```

Figure 2: A buffer overflow vulnerability.

In lines 1-5, *read_and_echo*() reads some user input and saves it in a local buffer *buf*. As the C library function *gets*() does not check the capacity of *buf*, there is a buffer overflow vulnerability in line 3. In addition, the C library function *printf*() invoked in line 4 also contains an information disclosure vulnerability [9] (exploited in a proof-of-concept attack in Section 4). By inputting more data than *buf* can hold, an attacker can corrupt the return address of *read_and_echo*() on the call stack with one of her choosing. When *read_and_echo*() returns, the control flow will be hijacked.

2.2 Return Oriented Programming

Figure 3 illustrates a ROP attack [42], where three ROP gadgets, A, B and C, are chained to compute `*target = num`. By exploiting a buffer overflow vulnerability (Figure 2), an attacker can replace the return address of *read_and_echo*() with the address of gadget A. Once this victim function returns, gadgets A, B and C will be triggered one by one, resulting in a control-flow hijacking attack.



```
05 mov %rsi,(%rdi)
06 ret
```

```
03 pop %rsi
04 ret
```

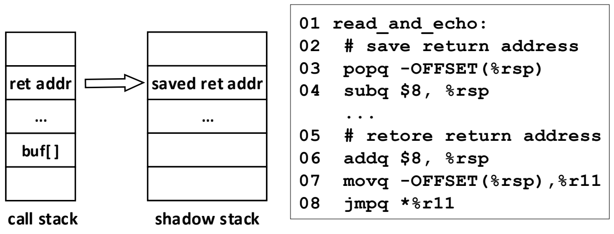
```
01 pop %rdi
02 ret
```

Figure 3: A ROP attack (by exploiting, say, the buffer overflow vulnerability in *read_and_echo*() of Figure 2), where %rsi contains the value of `num` and %rdi the value of `target`.

ROP is Turing complete [42]. As a special case, when a gadget begins at the entry of a function, we have a so-called *return-into-libc attack* [5, 42]. The C library function *system("/bin/sh")* is thus a popular choice for hijacking the victim by launching an interactive shell (in a proof-of-concept attack in Section 4).

2.3 The Traditional Shadow Stack

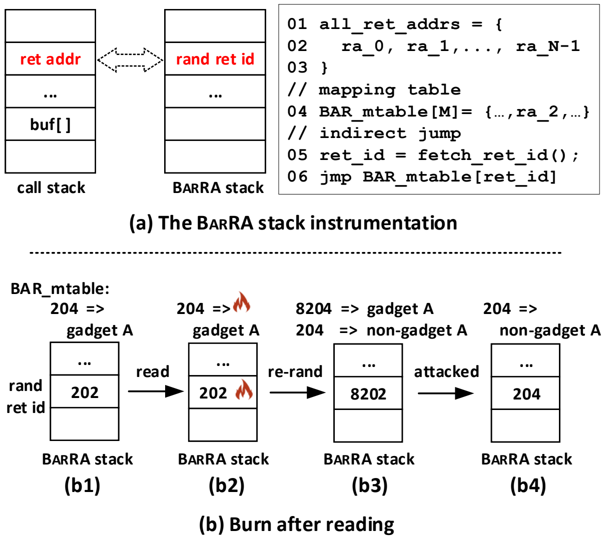
To mitigate a ROP attack illustrated in Figure 3, a shadow stack has traditionally been used to hide the return addresses on the call stack. To prevent the return address of *read_and_echo*() from being overwritten, as shown in Figure 4, the return address of *read_and_echo*() is saved in the shadow stack at the offset $-OFFSET + 8 + (\%rsp)$ (lines 3-4) on entering the function and restored (from the same location) on leaving the function (lines 6-8). By convention, %rsp is the standard stack pointer pointing to the top of the call stack. For simplicity, the shadow stack is assumed to have the same size as the call stack rather than as a FILO stack to save space [6, 21].



```
01 read_and_echo:
02  # save return address
03  popq -OFFSET(%rsp)
04  subq $8, %rsp
    ...
05  # retore return address
06  addq $8, %rsp
07  movq -OFFSET(%rsp),%r11
08  jmpq *%r11
```

Figure 4: The traditional shadow stack mechanism for protecting the return address of *read_and_echo*() in Figure 2.

As shown in Figure 4, the shadow stack usually appears below the call stack at a fixed offset, *OFFSET*, which is generated randomly at the beginning of program execution and then hidden subsequently, say, in a read-only code section. This mechanism, which relies on hiding the location of the shadow stack, is vulnerable to information disclosure and side-channel attacks [9, 16, 18, 36].



```
01 all_ret_addrs = {
02   ra_0, ra_1,..., ra_N-1
03 }
// mapping table
04 BAR_mtable[M]= {…,ra_2,…}
// indirect jump
05 ret_id = fetch_ret_id();
06 jmp BAR_mtable[ret_id]
```

(a) The BarRA stack instrumentation

(b) Burn after reading

Figure 5: The burn-after-reading shadow stack for protecting the return address of *read_and_echo*() in Figure 2 against the ROP attack (with gadgets A – C) illustrated in Figure 3.

As discussed already in Section 1, several complementary or orthogonal mitigation techniques are ineffective: ASLR [47] (which is still vulnerable to information disclosure and side-channel attacks [9, 16, 18, 36]), CFI [7, 15] (which still has a large attack surface), and runtime rerandomization on concrete return addresses [26] (which is too expensive in its pointer tracking operations to be applied frequently, and is thus also vulnerable).

2.4 The "Burn-After-Reading" Shadow Stack

Figure 5 illustrates our BarRA methodology, with the underlying shadow stack now referred to as the *BarRA stack*. Unlike the case for the shadow stack, even if the location of the BarRA stack is leaked, BarRA can still prevent the control flow from being hijacked by modifying the return address of *read_and_echo*() with a high probability, resulting in a significantly stronger security guarantee.

The key novelty of BarRA is to abstract a concrete return address with a randomized return id and store the abstract return address thus obtained in the BarRA stack. This one level of indirection makes it possible to apply continuous runtime rerandomization efficiently to the BarRA stack to enforce the burn-after-reading property (requiring all leaked return ids to be made obsolete).

Figure 5(a) illustrates how the BarRA stack works. By applying program analysis, all the N return addresses in the program are found and saved in *all_ret_addrs* (lines 1-3). During program execution, a table, named *BAR_mtable*, of size M, where $M \geqslant N$, maintains a mapping from return ids to their concrete return addresses (line 4). When a BarRA-protected function returns, its return id is fetched from the BarRA stack (line 5) and an indirect jump is made to its corresponding concrete return address (line 6).

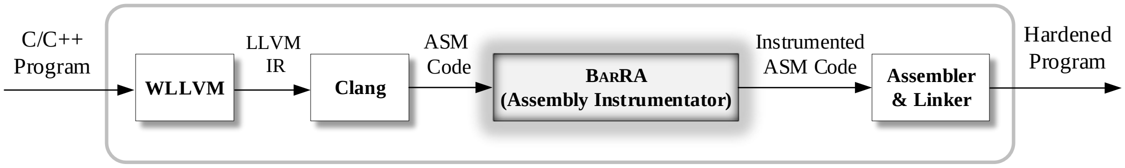


Figure 6: The workflow of BarRA.

Figure 5(b) illustrates how BarRA mitigates the ROP attack with gadgets A − C (Figure 3), where the attacker attempts to replace the return id of *read_and_echo*() with the return id of gadget A. Currently, the return id of *read_and_echo*() is 202 (Figure 5(b1)). By performing some information leakage attacks, the attacker has found the return id for gadget A to be 204. On detecting this information leak (Section 3.3), BarRA will enforce immediately the burn-after-reading property as illustrated in Figure 5(b2). This is done by rerandomizing the mapping *BAR_mtable*. Afterwards, as shown in Figure 5(b3), the return ids for *read_and_echo*() and gadget A have been changed to 8202 and 8204, respectively. Even if the attacker manages to replace the return id of *read_and_echo*() with 204 in the BarRA stack, as shown in Figure 5(b4), gadget A can no longer be executed (as its return id is now 8204)!

To recap from Section 1, BarRA has several salient properties:

- **Lightweight Runtime Rerandomization.** Rerandomizing *BAR_mtable* can be done simply by modifying a randomly generated offset added to all the return ids (modulo M). This requires the return ids in the BarRA stack to be updated, in the order of microseconds instead of seconds as required by RuntimeASLR (due to its expensive pointer tracking) [26], delivering a six-orders-of-magnitude speedup.
- **Low Instrumentation Overheads.** BarRA exhibits comparable instrumentation overheads as the shadow stack mechanism.
- **Strong Security Guarantees.** As the maximum number of return ids is M (the size of *BAR_mtable*), the chance of guessing correctly the return id for gadget A to be 8204 in Figure 5(b4) is only $\frac{1}{M}$. If we use 8MB of memory (on a 64-bit platform) to implement *BAR_mtable* (where $M \gg N$ is possible), an attacker's success rate is only $\frac{1}{2^{20}}$. Finally, unlike RuntimeASLR [26], BarRA integrates CFI (by limiting all gadgets to be within *all_ret_addrs* in Figure 5(a)) with the shadow stack and runtime rerandomization.

3 BARRA: DESIGN AND IMPLEMENTATION

We focus on protecting return addresses by adopting the same threat model as before [1, 21, 47, 48, 52**?** , 53]. The attacker can read any readable memory or write any writable memory by exploiting existing vulnerabilities in order to hijack the control flow.

We have designed and implemented BarRA in the LLVM compiler tool chain, as shown in Figure 6. Given a C/C++ program, its source files are compiled and linked by the LLVM tool chain into a single LLVM-IR (known as bitcode). We use a compiler wrapper, WLLVM [39], to build a whole-program LLVM bitcode file. The BarRA assembly instrumentator, which is added in this paper, generates instrumented assembly code that is amenable to lightweight runtime rerandomization on abstract return addresses. Finally, the instrumented assembly code is assembled and linked into a hardened binary by the assembler and linker, respectively.

As is standard, every function is assumed to have at most one return instruction (with jumps added where appropriate).

Below we use an example given in Figure 7 to illustrate the instrumentation added by BarRA. For the motivating example repeated in Figure 7(a), BarRA maintains the (unused) concrete return addresses in the call stack (for compatibility reasons) (Figure 7(b)) and their abstract return ids in the BarRA stack (Figure 7(c)). Figure 7(d) (Figure 7(e)) gives the instrumented code (data) section.

3.1 Data Instrumentation

Figure 7(e) lists an instrumented data section added, consisting of a read-only table *all_ret_addrs* containing all the return addresses in the program (lines 24-34), a randomly generated value *BAR_randval* (lines 35-42), and our mapping table *BAR_mtable* (lines 43-50). Both *BAR_randval* and *BAR_mtable* reside in the .BSS sections, and thus take no actual space in the object file.

In this example, *all_ret_addrs* of size $N = 5$ contains five return addresses. *BAR_randval*, which is a 8-byte value, is stored in a 4096-byte page (on page-aligned boundaries) such that the entire page can be set as read-only after *BAR_randval* has been generated after each round of runtime rerandomization. *BAR_mtable* of size M, where $M \geqslant N$, will be initialized at load time, such that $\forall\ 0 \leqslant i < N : BAR\_mtable[i] = all\_ret\_addrs[i]$ and $\forall\ N \leqslant i < M : BAR\_mtable[i]$ = address of a ROP catcher. When an attacker tampers with the BarRA stack with a stale return id mapped into the ROP catcher, a warning message can be issued.

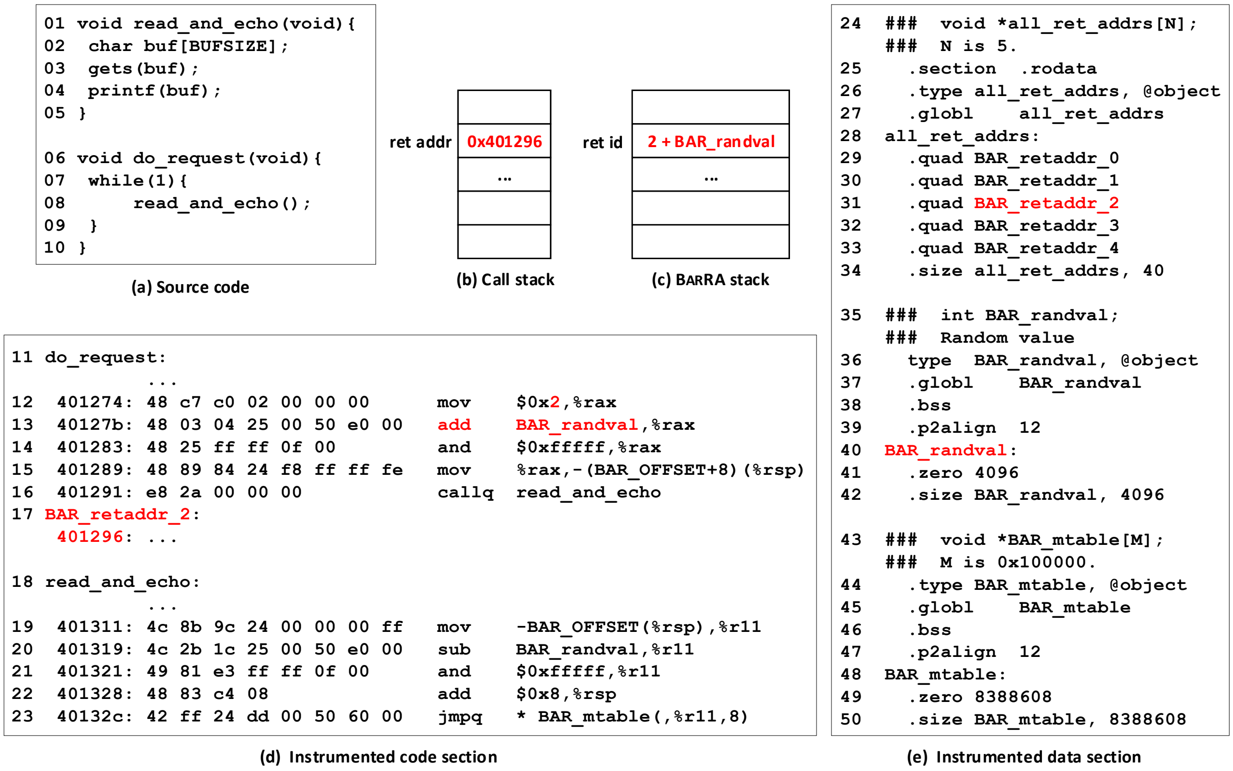
In our approach, *BAR_mtable* is disclosed to the attacker. However, on detecting information leaks (or at the program startup), every return id, i, will be changed randomly to $(i + BAR\_randval)$ mod M, so that the burn-after-reading property is enforced.

To provide strong security guarantees, *BAR_mtable* should be reasonably large. It is suggested to allocate 8MB of memory (on a 64-bit platform) for *BAR_mtable* so that it can hold up to $M = 0x100000$ (i.e., 1M) return addresses. This way, an attacker's success rate for guessing the return id of a gadget correctly is only $\frac{1}{M} = \frac{1}{2^{20}}$.

3.2 Code Instrumentation

Figure 7(d) shows how to instrument a call instruction and its corresponding return instruction. We need to add instrumentation code before the call for *read_and_echo*() (but after all its parameter-pass instructions, if any) in *do_request*() (line 16). We also add instrumentation code to replace the return instruction (not shown explicitly) in *read_and_echo*(), whose (concrete) return address is *BAR_retaddr_2*, i.e., $0x401296$ and (abstract) return id is 2.

Unlike the shadow stack mechanism that instruments a call instruction by adding its instrumentation code at the beginning of all the callee functions (Figure 4), BarRA instruments all the calls separately in order to also protect their return edges using CFI [1].



```
01 void read_and_echo(void){
02   char buf[BUFSIZE];
03   gets(buf);
04   printf(buf);
05 }

06 void do_request(void){
07   while(1){
08         read_and_echo();
09   }
10 }
```

(a) Source code

(b) Call stack

(c) BarRA stack

```
11 do_request:
        ...
12  401274: 48 c7 c0 02 00 00 00        mov    $0x2,%rax
13  40127b: 48 03 04 25 00 50 e0 00     add    BAR_randval,%rax
14  401283: 48 25 ff ff 0f 00           and    $0xfffff,%rax
15  401289: 48 89 84 24 f8 ff ff fe     mov    %rax,-(BAR_OFFSET+8)(%rsp)
16  401291: e8 2a 00 00 00              callq  read_and_echo
17 BAR_retaddr_2:
    401296: ...

18 read_and_echo:
        ...
19  401311: 4c 8b 9c 24 00 00 00 ff     mov    -BAR_OFFSET(%rsp),%r11
20  401319: 4c 2b 1c 25 00 50 e0 00     sub    BAR_randval,%r11
21  401321: 49 81 e3 ff ff 0f 00        and    $0xfffff,%r11
22  401328: 48 83 c4 08                 add    $0x8,%rsp
23  40132c: 42 ff 24 dd 00 50 60 00     jmpq   * BAR_mtable(,%r11,8)
```

(d) Instrumented code section

```
24  ###  void *all_ret_addrs[N];
    ###  N is 5.
25    .section  .rodata
26    .type all_ret_addrs, @object
27    .globl    all_ret_addrs
28  all_ret_addrs:
29    .quad BAR_retaddr_0
30    .quad BAR_retaddr_1
31    .quad BAR_retaddr_2
32    .quad BAR_retaddr_3
33    .quad BAR_retaddr_4
34    .size all_ret_addrs, 40

35  ###  int BAR_randval;
    ###  Random value
36    type  BAR_randval, @object
37    .globl    BAR_randval
38    .bss
39    .p2align  12
40  BAR_randval:
41    .zero 4096
42    .size BAR_randval, 4096

43  ###  void *BAR_mtable[M];
    ###  M is 0x100000.
44    .type BAR_mtable, @object
45    .globl    BAR_mtable
46    .bss
47    .p2align  12
48  BAR_mtable:
49    .zero 8388608
50    .size BAR_mtable, 8388608
```

(e) Instrumented data section

Figure 7: BarRA's instrumentation illustrated for an example program.

3.2.1 *Call Instructions.*

For the call to *read_and_echo*(), our instrumentation code (lines 12-15) inserts its randomized return id (2 + *BAR_randval*) mod *M* into the BarRA stack at its location −*BAR_OFFSET*−8+(*rsp*), where the modulo operation is realized in line 14. Here, (*rsp*) − 8 points to the return address *BAR_retaddr_2* on the call stack. As in the case of the shadow stack (Section 2.3), the BarRA stack appears below the call stack at a fixed distance of *BAR_OFFSET*. In line 16, *BAR_retaddr_2* will still be pushed into the call stack even it is not used (for compatibility reasons).

3.2.2 *Return Instructions.*

To replace the return instruction in *read_and_echo*(), we rely on some instrumentation code again (lines 19-22). In lines 19-21, we retrieve its return id saved after undoing the rerandomization as (2 + *BAR_randval*) (line 19) − *BAR_randval* (line 20) mod *M* (line 21) ≡ 2 from the BarRA stack. In line 22, we adjust *rsp* by skipping the (unused) return address *BAR_retaddr_2* in the call stack for compatibility reasons.

3.3 Runtime Rerandomization

On detecting an information leak (or at the program startup), BarRA will start a new round of runtime rerandomization (for the abstract return addresses), by invoking rerandomize() in Figure 8. Instead of rerandomizing *BAR_mtable* directly, we achieve the same effect (as validated easily by inspecting lines 12-15 and 19-23 in Figure 7(d)) more efficiently by modifying BAR_randval randomly.

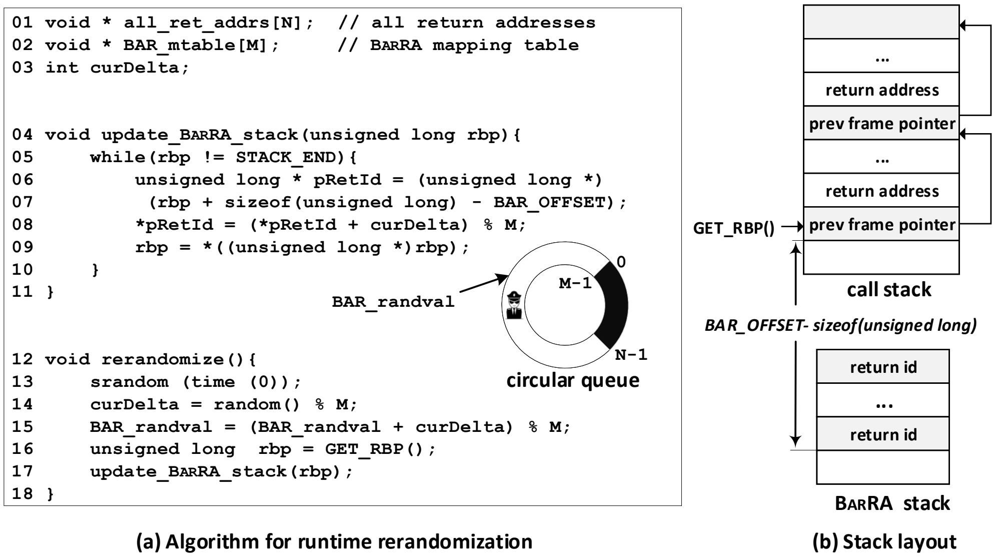
To randomize BAR_randval, as shown in Figure 8(a), we increment it by a randomly generated offset, *curDelta* (lines 13-15). We then call *update_BarRA _stack(rbp)* to add this offset to the return ids in the BarRA stack (lines 16-17). In line 16, *rbp* = *GET_RBP*() is initialized to point to the beginning of the list of the frame pointers (rbps) saved on the call stack, as illustrated in Figure 8(b). By traversing this list (lines 5-10), we can locate the return ids for all the callers stored in the corresponding BarRA stack.

There are general approaches to detecting information leaks in a program [19, 55]. In our evaluation, we monitor whether some input/output functions such as *gets*() in Figure 7 are called (as in [4]) and invoke *rerandomize*() as soon as this has happened. Specifically, every input/output function can be hooked so that *rerandomize*() will be always executed just before it. On arriving at the server (where a BarRA-protected program is running) from a remote attacker, a malicious packet will be handled by such hooked functions, resulting in burn-after-reading (via rerandomization).

3.4 An Example

We revisit our motivating example by refining Figure 5(b) to Figure 9 to see how BarRA mitigates the ROP attack in Figure 3, in which the attacker attempts to replace the return id of *read_and_echo*() with the return id of gadget A in the BarRA stack.

At time *t*0, we have two return ids, 2 for *read_and_echo*() and 4 for gadget A. At the program startup *t*1, BAR_randval = 200 is generated. Just before *read_and_echo*() is called (lines 12-15 in



```
void * all_ret_addrs[N];  // all return addresses
void * BAR_mtable[M];     // BarRA mapping table
int curDelta;


void update_BarRA_stack(unsigned long rbp){
    while(rbp != STACK_END){
        unsigned long * pRetId = (unsigned long *)
         (rbp + sizeof(unsigned long) - BAR_OFFSET);
        *pRetId = (*pRetId + curDelta) % M;
        rbp = *((unsigned long *)rbp);
    }
}


void rerandomize(){
    srandom (time (0));
    curDelta = random() % M;
    BAR_randval = (BAR_randval + curDelta) % M;
    unsigned long  rbp = GET_RBP();
    update_BarRA_stack(rbp);
}
```

(a) Algorithm for runtime rerandomization

(b) Stack layout

Figure 8: BarRA's runtime rerandomization (with the police icon representing the return ids mapped to a ROP catcher).

	init		read			attacked	return	
	t0	t1	t2	t3	t4	t5	t6	t7
curDelta	0	200	200		8000	8000	8000	
BAR_randval	0	200	200	🔥	8200	8200	8200	👮
return id	2	202	202	🔥	8202	204	(204 - 8200)%M	
first ROP gadget id	4	204	204		8204	8204	8204	

Figure 9: BarRA's mitigation for the ROP attack in Figure 3 by protecting the return id of *read_and_echo*() in Figure 7.

Figure 7(d)), its randomized return id 202 is inserted into the BarRA stack. The randomized return id for gadget A has been changed to 204, which is discovered by the attacker at *t*2. To enforce burn-after-reading, BarRA calls rerandomize() at *t*3, so that BAR_randval = *curDelta* + 200 = 8000 + 200 = 8200. At *t*4, the randomized return id of *read_and_echo*() on the BarRA stack has been changed to 8202, and the return id of gadget A has been re-randomized to 8204. With the attack coming at *t*5, the randomized id 8202 on the BarRA stack will be overwritten with a malicious but stale gadget id 204. When the attacked function returns, the instrumented code in lines 19-22 in Figure 7(d) will generate a return id (204 − 8200)%*M*, which is not gadget A. If (204 − 8200)%*M* represents our ROP catcher, then the attack will be flagged at *t*7 (marked by the police icon).

4 EVALUATION

We have implemented BarRA in C/C++ in the LLVM compiler tool chain. Currently, BarRA supports both 32-bit and 64-bit x86 assembly code.

To ensure that BarRA is compatible with closed-source, i.e., unprotected binaries, we have also implemented a simple static analysis on top of SVF [44], an open-source pointer analysis framework, to identify the functions in a software application that may be called from closed-source binaries. These functions will be protected by the traditional shadow stack mechanism as their call instructions cannot be instrumented. In future work, we will consider binary instrumentation [28, 33] and disassembly [2, 30] to provide full BarRA-protection for all the functions in the program.

Our evaluation demonstrates the efficiency and effectiveness of BarRA in protecting return addresses in real-world applications (with the shadow stack as the baseline). We have selected all the 19 C/C++ programs in SPEC CPU2006 (totaling 2,047,447 LOC), including 67,855 functions and 339,983 call instructions (Table 3).

We have also developed a proof-of-concept attack, which works by exploiting some format string vulnerabilities, such as CVE-2019-7715 and CVE-2019-7712 on the CVE website [13]. We show that under some information leakage attacks, the shadow stack is always vulnerable but BarRA is substantially more secure.

We address the following three research questions (RQs):

- **RQ1.** Is BarRA's runtime rerandomization lightweight?
- **RQ2.** Does BarRA have low instrumentation overheads?
- **RQ3.** Is BarRA effective in protecting return addresses in the presence of information disclosure vulnerabilities?

Our platform consists of a 3.20 GHz Intel Xeon(R) E5-1660 v4 CPU with 256 GB memory, running the (64-bit) Ubuntu OS. All the SPEC CPU2006 programs are compiled under the optimization flag "-O2". When running a program, the call stack used is 8MB (by default) and its corresponding shadow/BarRA stack is also 8MB. The time measurement for each metric is the average of 5 runs.

To enforce the burn-after-reading property, BarRA uses 8MB of memory to implement its *BAR_mtable* as illustrated in Figure 7. However, this is not needed by the traditional shadow stack.

4.1 RQ1: Lightweight Rerandomization

According to rerandomize() in Figure 8, the time spent in each round of runtime rerandomization depends on the depth of the call stack. By reading */proc/pid/stack* for all the 19 SPEC CPU benchmarks, the depths of their call stacks are found to be all under 256. To cover more cases, we have written a range of test programs with

Table 1: The times spent for BarRA to perform runtime rerandomization under different call stack depths.

Call Stack Depth	Time (microseconds)
64	2
128	2
256	3
512	5
1024	8
2048	17
4096	33
8192	63

Table 2: Comparing RuntimeASLR and BarRA.

	RuntimeASLR	BarRA
Address Space	Concrete	Abstract
Randomization Timing	fork()	Any Time
Coverage	All Pointers	Return Addresses
Pointer tracking	✓	×
Burn-after-Reading	×	✓
Overhead	Seconds	Microseconds

different call stack depths. The runtime rerandomization overheads under different call stack depths are listed in Table 1. It is worth emphasizing that no more than 3 microseconds are needed when the call stack depth is below 256. Elsewhere [4], the average call stack depth observed is much smaller. In general, the time taken by rerandomize() is linearly proportional to the depth of the call stack.

Table 2 summarizes the major differences between BarRA (operating on an abstract address space) and RuntimeASLR [26] (a state-of-the-art runtime rerandomizer operating on a concrete address space), as already discussed in Section 2. RuntimeASLR rerandomizes all the pointers in the program. In contrast, BarRA rerandomizes only the return addresses (i.e., backward edges) while leaving the protection of the forward edges to CFI [1]. By operating on a concrete address space, RuntimeASLR must perform expensive and difficult pointer tracking. Due to its excessive overheads (e.g., 217 seconds for *soplex*), RuntimeASLR opts to apply runtime rerandomization to a child process only at the time of *fork()*, thus failing to enforce the burn-after-reading property in the presence of frequent information leakage attacks. In contrast, BarRA is lightweight, making it suitable to enforce the burn-after-reading property whenever some information leaks are detected.

In summary, BarRA performs runtime rerandomization in the order of microseconds while RuntimeASLR operates in the order of seconds. For the call stacks containing less than 256 calls, BarRA turns out to be six-orders-of-magnitude faster (Table 1).

4.2 RQ2: Low Instrumentation Overheads

Table 3 lists the statistics for the 19 C/C++ SPEC benchmarks. For each benchmark, #LOC, #Text, #Funs and #Calls give its source code size, its binary code size, the number of its functions, and the number of its calls, respectively. Under "Inst Overheads" (Columns 6-7), we see the instrumentation overheads introduced by the traditional shadow stack and BarRA. Under "Code Size Increases" (Columns 8-9), we see the code size increases under two approaches.

Below we compare both approaches in terms of their instrumentation overheads introduced and code size expansion incurred.

4.2.1 Instrumentation Overhead. Despite significantly stronger guarantees provided (Section 4.3), BarRA exhibits comparable instrumentation overheads as the shadow stack, 5.38% for the shadow stack vs. 6.09% for BarRA (on average), as revealed in Columns 6-7. These results correlate well with a similar number of instrumentation instructions executed under the shadow stack (Figure 4) and BarRA (Figure 7). To instrument a call, BarRA introduces a memory read (for *BAR_randval*) in line 13 and a memory write in line 15 (Figure 7). In contrast, the shadow stack adds the instrumentation code at the beginning of each of its callee functions (Figure 4). Its *pop* instruction (line 3) consists of essentially a read on the call stack and a write on the shadow stack. As the page that contains *BAR_randval* is accessed frequently, its cache hit rate is expected to be high, reducing its memory read overhead. In addition, BarRA uses two additional non-memory-access instructions (lines 12 and 14 in Figure 7), while the shadow stack uses only one (line 4 in Figure 4). A similar analysis applies to the instrumented code for a return instruction. Therefore, BarRA is expected to have similar instrumentation overheads as the shadow stack.

4.2.2 Code Size Expansion. In order to protect return addresses better than the shadow stack (Section 4.3), BarRA generates slightly larger binaries across the benchmarks, 6.15% for the shadow stack vs. 29.44% for BarRA (on average), as revealed in Columns 8-9. For a program, the extra code added by BarRA consists of (1) a table *all_ret_addrs* containing its return addresses (Figure 7(e)), (2) the instrumentation code for its call instructions (lines 12-15 in Figure 7(d)), and (3) the instrumentation code for its return instructions (lines 19-23 in Figure 7(d)). The total code size increase in (1) and (2) is proportional to the number of call instructions while the code size increase in (3) is proportional to the number of functions in the program. In contrast, the number of instrumentation instructions added by the shadow stack to a program is always proportional to the number of its functions (Figure 4). As a program has usually more calls than functions (Table 3), BarRA is expected to generate slightly larger binaries than the shadow stack, but in return for stronger security guarantees provided (as discussed below).

In our actual implementation, the three instrumentation instructions (lines 13-15 in Figure 7(d)) shared at all calls to a function are factorized and moved to the beginning of the function.

4.3 RQ3: Strong Security Guarantees

First of all, BarRA has made a program's attack surface substantially smaller than the shadow stack. Due to the CFI property enforced by BarRA, the attack surface is limited to the set of potential gadgets falling into *all_ret_addrs* (Figure 7(e)). For the shadow stack, however, any gadget found in the program can be exploited.

In Figure 10, we demonstrate via a proof-of-concept attack that the shadow stack is always vulnerable but BarRA is significantly more secure in the presence of information leakage attacks. We assume a 32-bit platform with 4-byte addresses. In Figure 10(a), we have implemented a multiple-process echo server on Ubuntu

Table 3: The statistics for the 19 C/C++ SPEC benchmarks.

Benchmark	#LOC	#Text	#Funs	#Calls	Inst Overheads		Code Size Increases	
					Shadow Stack	BarRA	Shadow Stack	BarRA
bzip2	8381	83271	100	425	0.77%	3.74%	4.56%	18.07%
gcc	517621	4434399	5572	52763	3.16%	4.14%	3.13%	24.42%
gobmk	197303	1613664	2679	9980	5.65%	9.30%	1.80%	8.21%
h264ref	51752	692436	590	3543	1.93%	4.63%	2.32%	11.57%
hmmer	35992	345292	538	4086	0.57%	0.38%	4.65%	23.15%
lbm	1155	23518	19	71	-0.37%	5.17%	0.31%	13.75%
libquantum	4357	43429	115	556	0.54%	1.62%	8.82%	43.46%
mcf	2685	17381	24	79	4.35%	7.02%	18.36%	36.61%
milc	15042	141659	235	1619	1.07%	1.47%	5.64%	25.26%
perlbench	168274	1419350	1870	15334	3.34%	2.63%	3.41%	22.99%
sjeng	13847	155749	144	1361	4.04%	5.57%	2.56%	17.64%
sphinx	25104	205120	369	2753	0.20%	0.59%	3.86%	28.78%
astar	5842	49896	153	665	3.93%	4.44%	7.52%	36.92%
dealII	198642	4165302	19234	93380	29.81%	20.26%	12.07%	57.62%
namd	3188	477495	140	1497	0.68%	0.54%	1.72%	5.99%
omnetpp	48040	800247	2765	21041	15.69%	17.56%	9.08%	49.38%
povray	155163	1131443	2023	15114	17.14%	19.05%	4.64%	28.53%
soplex	41428	442572	1542	9867	8.63%	7.55%	9.14%	51.09%
Xalan	553631	5516159	29743	105849	1.12%	0.07%	13.31%	55.99%
Total	2,047,447	21,758,382	67,855	339,983				
Average					5.38%	6.09%	6.15%	29.44%

Table 4: Read and write primitives by exploiting format strings in printf().

Read/Write Primitive	Format String	Example	Semantics
Absolute write primitive	%ku%hn	printf("%65535u%hn", val, &cnt)	Write two bytes, 0xFFFF (65535), to &cnt
Absolute read primitive	%s	printf("%s", &str)	Read the content at &str
Relative read primitive	%08x	printf("%08x%08x%08x%08x%08x")	Read the first five anonymous arguments
Relative read primitive	%k$u	printf("%3$u")	Read the 3rd anonymous argument

(lines 1-23), which receives a message from a client and sends the message back to the client via a socket connection. The code for creating socket connections has been elided. The system call *fork()* in line 13 creates a child process and *dup*2() in line 16 redirects the standard I/O of the server to the socket connection, so that *gets*() in line 3 will read a message from a remote client and *printf()* in line 4 will send it back to the client. The source code in lines 1-10 is the same as that in the motivating example in Figure 2.

The attacker will exploit the format string vulnerability in *printf()* in line 4. Some read and write primitives at his/her disposal are listed in Table 4. As illustrated in Figure 10(b), there are two internal pointers maintained in *printf()*: *fmt* initially points to the beginning of the format string (line 24) and *pArg* is initialized to point to the first anonymous argument on the stack (line 25). For a variadic function like *printf()*, its number of anonymous arguments is determined by parsing the format string pointed by *fmt* (lines 26-39). All the format specifiers (substrings starting with '%') are handled in the normal manner. In the special case when the formal specifier is "%08x" (lines 28-32), the anonymous argument pointed by *pArg* is printed as a sequence of 8 characters (with leading 0's added if necessary). Afterwards, *pArg* is made to point to next anonymous argument. Once an attacker has controlled the format string, the attacker can inspect the content in the call stack (by printing some "anonymous arguments" never passed explicitly to *printf()*).

Let us explain first how this echo server, protected by the shadow stack, will be smashed by a formal string attack launched from *printf()* in line 4. We then discuss briefly why BarRA provides significantly stronger security guarantees under the same attack.

4.3.1 *Attacking the Traditional Shadow Stack.*

In Figure 10(c), a shadow-stack-hardened echo server *server_SS* is listening on port 9999. Due to some information disclose attacks (by experimenting with different probing format strings sent to the server), the attacker has succeeded in obtaining the following valuable information about the echo server: the return address of *read_and_echo*() is 0x08048ea7 stored at 0xffc2c7cc in the call stack and also at 0xfec2c7d0 in the shadow stack, the C library function *system*() resides at 0xf7e26da0, and the string *"/bin/sh"* starts at 0xf7f47a0b.

The attacker is now in action. In Figure 10(e), the attacker sends a packet that contains a malicious format string to *server_SS*. For

Table 5: BarRA's three-in-one approach for protecting return addresses.

Mechanism	Strengths	Weaknesses
① Shadow Stack	Reducing Buffer Overflow Attacks	Vulnerable to Information Leakage Attacks
② Control flow integrity (Backward)	Reducing Available ROP Gadgets	Still Permitting Illegal Return Addresses
③ Runtime Rerandomization (Concrete)	Alleviating Information Leakage Attacks	Problematic Pointer Tracking
BarRA (① + ② +③ (Abstract))	All the Above but without Pointer Tracking	Probabilistic Protection

clarity, we have split it into multiple lines (lines 48-54). The objective, as indicated in the inline comments (lines 45-47), is to (1) overwrite the return address of *read_and_echo*() in the shadow stack by 0xf7e26da0 (the address of *system*()), as shown in Figure 10(g), and (2) restore the return address of *read_and_echo*() as 0x08048ea7 in the call stack (for a graceful exit without leaving any trace) and inject 0xf7f47a0b (the address of *"/bin/sh"*) into the call stack (as the argument of *system*()), as shown in Figure 10(h).

When *read_and_echo*() returns, *system*(*"/bin/sh"*) will be executed. The format string vulnerability at *printf()* in line 4 on the server has resulted in a control-flow hijack, leading to an interactive shell environment that the attacker can control remotely, as shown in lines 55-56 of Figure 10(e). In line 55, the attacker enters a command *date*, the reply (in line 56) from the *server_SS* indicates that the attacker has succeeded in hijacking the remote server.

Below we explain briefly how the format string vulnerability at *printf()* in line 4 is exploited to launch this attack. In line 3, *gets*() reads the malicious format string prepared in lines 48-54 by the attacker (Figure 10(e)) into *buf* stored in the call stack depicted in Figure 10(g). Just before *printf()* starts its execution, its two internal pointers, *fmt* and pArg, point to this format string and first anonymous argument on the call stack, respectively. As there may be some gap between *buf* and *fmt* (due to, e.g., register spilling), the attacker may have to experiment with the right number of "%08x"'s used in the malicious format string.

Let us focus on describing how 0xf7e26da0 (the address of *system*()) is written into the shadow shadow as a fake return address for *read_and_echo*(), as illustrated in Figure 10(g). The format string parser used internally by *printf()* parses the first 16×3 regular characters in lines 48-50, which are printed directly in lines 35-38. Then in line 51, a sequence of five format specifiers "%0x8" are encountered. After having handled all these five formal specifiers in lines 28-32 (with each printed as a sequence of 8 chars), *pArg* will be lifted to point to *buf.* At this point, *printf()* has printed $16\times3+8\times5$ characters after having processed lines 48-51. Next, *printf()* parses "%27976u" in line 52, causing the current anonymous argument on the call stack, i.e., 0x41414141 to be printed as 27976 characters. On encountering the first "%hn" in line 52, *printf()* has printed $16\times3+8\times5+27976$, i.e., 0x6da0 characters, which are the lower two bytes of the address of the function *system()*, i.e., 0xf7e26da0. This value will be written to the shadow stack at 0xfec2c7d0 pointed by the current anonymous argument on the call stack. After having handled the remaining string "%35394u%hn" in line 52, the higher two bytes of the address of *system()*. i.e., 0xf7e2 will be written to the shadow stack at 0xfec2c7d2. By parsing lines 53-54 (the remaining of the format string) in a similar manner, the return address of *read_and_echo*(), i.e., 0x08048ea7 and the address of *"/bin/sh"* will be written into the call stack, as shown in Figure 10(h).

4.3.2 *Attacking the Burn-After-Reading BarRA Stack.*

Our BarRA-hardened echo server, *server*_BarRA, listening on port 8080 as shown in Figure 10(d), will be significantly more secure. The attacker can obviously attempt to smash the server in a similar way by exploiting a format string vulnerability as shown in Figure 10(f). According to line 65, the attacker would like to replace the return id of *read_and_echo*() currently stored in the BarRA stack by 0x000000cc, i.e., 204 in line 65 (as per the comment in line 58) so that the corresponding gadget (say, gadget A in Figure 3) will be executed when *read_and_echo*() returns.

In order to discover this gadget, some information leakage attacks must be made. As illustrated earlier in Figure 9, this will trigger BarRA's *rerandomize()* to rerandomize *BAR_mtable* by changing 204 randomly into, say, 8204, defeating effectively the attack as shown in Figure 10(d). In our evaluation, *BAR_mtable* is of size $M = 2^{20}$. The chance for guessing a gadget id correctly is very low, i.e., $\frac{1}{M} = \frac{1}{2^{20}}$ only.

Due to the enforcement of CFI by BarRA, *system*() may no longer be used as a gadget, as its address is not in *BAR_mtable*.

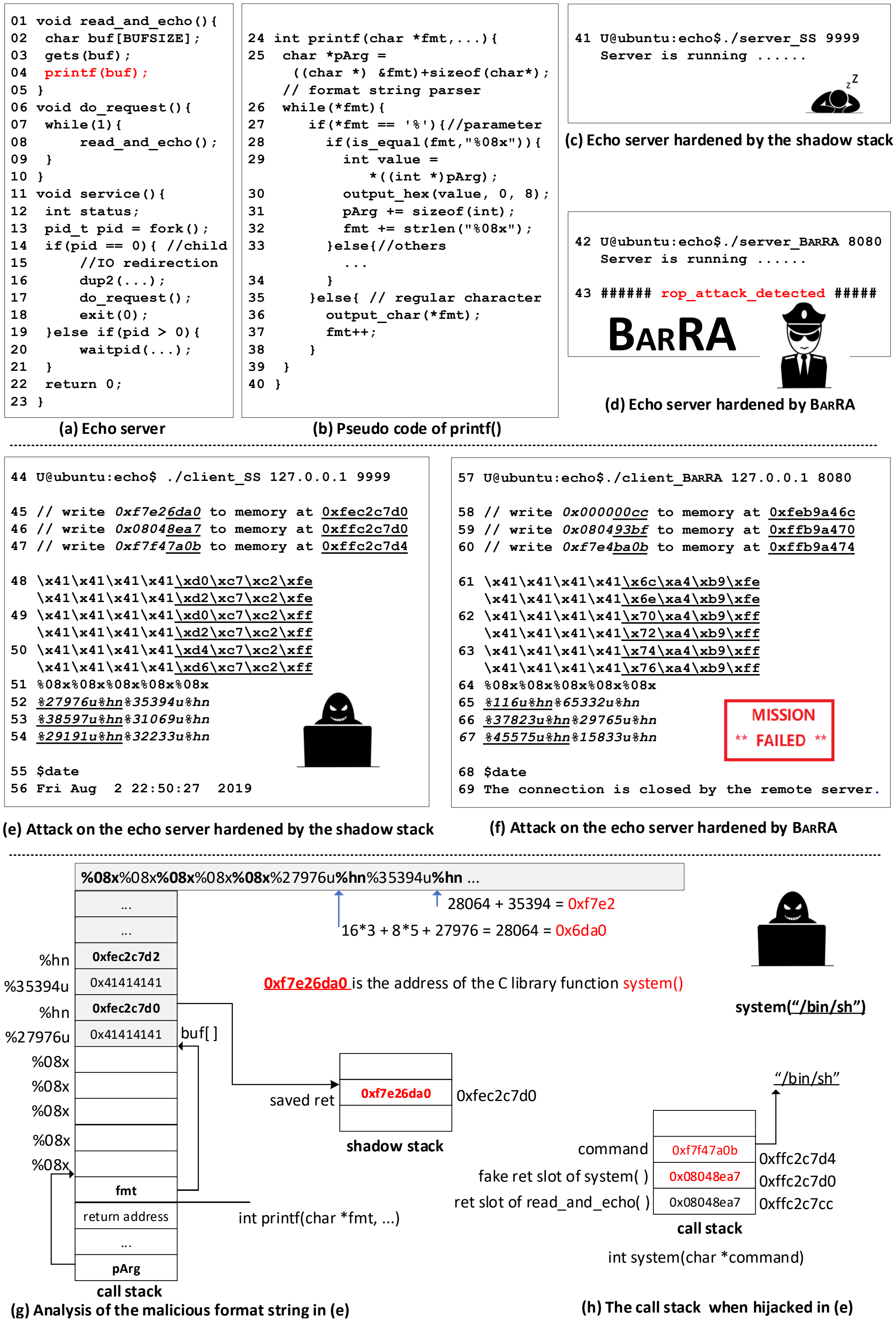
As shown in Figure 7(e), *BAR_mtable* of size M contains only N return addresses in its first N slots, where $M \geqslant N$. All the other slots are mapped to a ROP catcher. The possibility for the attacker to be caught immediately by our ROP catcher is given by $\frac{M-N}{M}$.

If we are to forgo this instant attack-catching capability (as demonstrated in Figure 10(d)), the slots filled with our ROP catcher can be omitted to save memory. In this case, only a large virtual (rather than physical) memory region needs to be allocated to *BAR_mtable* to achieve a high entropy. Only when part of the virtual memory region (containing no return addresses) is accessed (attacked) will we need to allocate its corresponding physical pages.

4.3.3 *Discussions.*

Therefore, BarRA represents a three-in-one solution that combines naturally shadow stack, control flow integrity and runtime rerandomization together to protect return addresses against control-flow hijacks, as summarized now in Table 5.

In the absence of information leakage, BarRA provides the same level of security guarantees as the traditional shadow stack mechanism in mitigating buffer overflow attacks. In the presence of information leakage, however, the shadow stack becomes bypassable but BarRA will still provide strong security guarantees albeit probabilistically. There are two reasons. First, BarRA restricts a gadget to start only with an address maintained in *all_ret_addrs* (Figure 7(e)) due to the CFI thus enforced. In contrast, the shadow stack mechanism allows a gadget to start from any address. Second, BarRA invokes immediately rerandomize() (Figure 8) to modify the mapping maintained in *BAR_mtable* (on detecting an information leakage), reducing significantly the chances for the attacker to hijack the control flow successfully (as shown in Figure 5).



```
void read_and_echo(){
 char buf[BUFSIZE];
 gets(buf);
 printf(buf);
}
void do_request(){
 while(1){
     read_and_echo();
 }
}
void service(){
 int status;
 pid_t pid = fork();
 if(pid == 0){ //child
     //IO redirection
     dup2(...);
     do_request();
     exit(0);
 }else if(pid > 0){
     waitpid(...);
 }
 return 0;
}
```

(a) Echo server

```
int printf(char *fmt,...){
char *pArg =
  ((char *) &fmt)+sizeof(char*);
// format string parser
while(*fmt){
   if(*fmt == '%'){//parameter
     if(is_equal(fmt,"%08x")){
       int value =
          *((int *)pArg);
       output_hex(value, 0, 8);
       pArg += sizeof(int);
       fmt += strlen("%08x");
     }else{//others
       ...
     }
   }else{ // regular character
     output_char(*fmt);
     fmt++;
   }
 }
}
```

(b) Pseudo code of printf()

```
U@ubuntu:echo$./server_SS 9999
   Server is running ......
```

(c) Echo server hardened by the shadow stack

```
U@ubuntu:echo$./server_BARRA 8080
   Server is running ......

###### rop_attack_detected #####
```

(d) Echo server hardened by BARRA

```
U@ubuntu:echo$ ./client_SS 127.0.0.1 9999

// write 0xf7e26da0 to memory at 0xfec2c7d0
// write 0x08048ea7 to memory at 0xffc2c7d0
// write 0xf7f47a0b to memory at 0xffc2c7d4

\x41\x41\x41\x41\xd0\xc7\xc2\xfe
\x41\x41\x41\x41\xd2\xc7\xc2\xfe
\x41\x41\x41\x41\xd0\xc7\xc2\xff
\x41\x41\x41\x41\xd2\xc7\xc2\xff
\x41\x41\x41\x41\xd4\xc7\xc2\xff
\x41\x41\x41\x41\xd6\xc7\xc2\xff
%08x%08x%08x%08x%08x
%27976u%hn%35394u%hn
%38597u%hn%31069u%hn
%29191u%hn%32233u%hn

$date
Fri Aug  2 22:50:27  2019
```

(e) Attack on the echo server hardened by the shadow stack

```
U@ubuntu:echo$./client_BARRA 127.0.0.1 8080

// write 0x000000cc to memory at 0xfeb9a46c
// write 0x080493bf to memory at 0xffb9a470
// write 0xf7e4ba0b to memory at 0xffb9a474

\x41\x41\x41\x41\x6c\xa4\xb9\xfe
\x41\x41\x41\x41\x6e\xa4\xb9\xfe
\x41\x41\x41\x41\x70\xa4\xb9\xff
\x41\x41\x41\x41\x72\xa4\xb9\xff
\x41\x41\x41\x41\x74\xa4\xb9\xff
\x41\x41\x41\x41\x76\xa4\xb9\xff
%08x%08x%08x%08x%08x
%116u%hn%65332u%hn
%37823u%hn%29765u%hn
%45575u%hn%15833u%hn

$date
The connection is closed by the remote server.
```

(f) Attack on the echo server hardened by BARRA

(g) Analysis of the malicious format string in (e)

(h) The call stack when hijacked in (e)

Figure 10: Echo server hardened by BARRA and shadow stack.

To launch a ROP attack by exploiting a vulnerability, the attacker needs to chain together a number of ROP gadgets in the same payload and then replace a legitimate return id with the return id of the first gadget in the chain, as illustrated in Figure 3. With runtime rerandomization in BarRA, the probability of guessing such a correct return id to use is extremely low. If we use $M = 8$MB of memory (on a 64-bit platform) to store *BAR_mtable* containing N return addresses, where $M \gg N$, the attacker's success rate in guessing a correct return id to use as a gadget is only $\frac{1}{M} = \frac{1}{2^{20}}$.

BarRA applies runtime rerandomization by using the hooked input/output functions as before [4]. By avoiding expensive pointer tracking as in RuntimeASLR [26], BarRA can start a new round of its microsecond-level rerandomization at any other time when needed. To further reduce memory accesses in the instrumented code, a dedicated register can be allocated for storing *BAR_randval*.

4.4 Limitations

To enforce burn-after-reading, and consequently, provide stronger security guarantees than the shadow stack, BarRA needs to consume extra memory for maintaining *BAR_mtable* to achieve a high entropy (Figure 7(e)). With 8MB (on a 64-bit platform), we can reduce an attacker's success rate to $\frac{1}{2^{20}}$ (as discussed above). In addition, BarRA also generates slightly larger binaries (Table 3).

One policy for handling a multiple-threaded program is to let every thread maintain a separate BarRA stack, protected by a thread-local version of *BAR_randval* (say, in the thread local storage *%fs:BAR_randval@tpoff* on x86). While avoiding the cross-thread synchronization issue, this simple solution does not guarantee that all thread-specific versions of BAR_randval are updated simultaneously, leading to potential cross-thread stack-smashing attacks [50]. A more secure solution would be to maintain only one single BAR_randval for all the threads and modify the thread dispatcher to suspend all the threads in the current process when runtime rerandomization occurs. We leave this to our future work.

Currently, we do not consider just-in-time code generation, which can already be handled by some CFI techniques [34]. In this case, BarRA can be extended to instrument the dynamically generated code and update *BAR_mtable* appropriately.

5 RELATED WORK

Low-level languages like C/C++ trade security guarantees for performance advantages. The absence of bounds-checking leads to memory errors [46] in C/C++ programs. StackGuard [11] inserts a canary in every stack frame and then tests whether the canary on the call stack has been corrupted or not when the corresponding function returns. Despite its lightweightness, StackGuard is not secure, as the attacker is still able to circumvent it by exploiting some information disclosure vulnerabilities [9, 40].

SafeStack, as a part of the code pointer integrity project [25], takes advantage of program transformation and information hiding to hide all the code pointers of a program in the safe region. The memory errors arising in the unsafe region do not compromise the safety guarantees made for safe region. Thus, the integrity of code pointers can be guaranteed. However, the safe region for a program is too large and thus vulnerable to side channel attacks [16, 36].

SoftBound [31] enforces spatial memory safety for C programs. By applying compile-time program transformations, SoftBound works by maintaining and reasoning about the metadata (base addresses and bounds) for all the pointers in the program. Later, SoftBound is augmented orthogonally to enforce also temporal memory safety [32]. To provide both types of memory safety, however, the combined instrumentation overhead is as high as 116%.

ASLR [47] randomizes the base addresses of code and data sections of a program at load time. In particular, ASR [17] performs fine-grained rerandomization for the Minix 3 microkernel. CCFIR [54] applies coarse-grained CFI and randomization for binary executables, but was shown later to be bypassable [15]. By exploiting information disclosure attacks, JIT Code Reuse [43] reduces the effectiveness of code randomization [20, 24, 37, 49]. Protection mechanisms that use secret memory regions to hide information, such as CPI [25] and ASLR-Guard [27], can also be bypassed [16, 18, 22]. In Readactor [12], all the code sections are hidden but the code pointers in data sections are still exploitable.

Runtime rerandomization that operates on a concrete address space requires expensive and difficult pointer tracking in a program [26]. For complex C/C++ programs, existing pointer-tracking techniques [10, 14, 41] are inadequate in discovering all kinds of pointers reliably. With the support of a customized compiler, TASR [4], which applies to C rather than C++, can rerandomize the code sections of a C program at runtime, but it does not work properly when function pointers are treated as data pointers. RuntimeASLR [26] has reduced its false positive rate to a negligible level, but still too costly (as discussed in Section 4.1).

In their seminal research on Control-Flow Integrity (CFI) [1], Abadi et al observed that the control flow graph of a program is an inherent property of the program and all runtime program execution paths should be constrained to be within the control flow graph. Their work has spurred a great deal of research on enforcing CFI [21, 35, 48, 52**?** , 53] and avoiding type-confusing errors [56**?**] in the past decade or so. To the best of our knowledge, CFI [1] has been applied to protect forward edges, i.e., indirect calls via function pointer and virtual calls (instead of backward edges, i.e., return addresses). The research on forward-edge CFI assumes usually that the shadow stack mechanism is used for enforcing backward-edge CFI [21, 48**?**]. However, the shadow stack is vulnerable to information leakage attacks, as demonstrated in our proof-of-concept attack (Section 4.3). In contrast, BarRA provides a significantly more secure mechanism for protecting return addresses, by enforcing burn-after-reading via lightweight runtime rerandomization on abstract return addresses.

6 CONCLUSION

In this paper, we introduce a novel shadow stack mechanism, BarRA, that applies continuous lightweight runtime rerandomization whenever some information leaks are detected, thereby enforcing the burn-after-reading property and making the traditional shadow mechanism significantly more secure. Enforcing burn-after-reading is essential for BarRA to mitigate information disclosure vulnerabilities effectively. Performing lightweight runtime rerandomization on abstract return addresses (with one level of indirection) is the new enabling technology proposed here.

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, NY, USA, 340–353.

[2] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Network and Distributed System Security Symposium*.

[3] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 255–270.

[4] 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, NY, USA, 268–279.

[5] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, New York, NY, USA, 30–40.

[6] 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*. 985–999.

[7] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium*. USENIX Association, Berkeley, CA, USA, 161–176.

[8] 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, NY, USA, 559–572.

[9] 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, NY, USA, 75–84.

[10] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. 2007. Effective Memory Protection Using Dynamic Tainting. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 284–292.

[11] 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 Conference on USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 1–16.

[12] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. 763–780.

[13] CVE. [n. d.]. Common Vulnerabilities and Exposures. https://cve.mitre.org/. Accessed Jan 24, 2020.

[14] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, New York, NY, USA, 482–493.

[15] 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 Conference on Security Symposium*. USENIX Association, Berkeley, CA, USA, 401–416.

[16] 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, San Jose, CA, USA, 781–796.

[17] 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 Conference on Security Symposium*. USENIX Association, Berkeley, CA, USA, 40–55.

[18] 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*.

[19] Salvatore Guarnieri and V Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code.. In *USENIX Security Symposium*, Vol. 10. 78–85.

[20] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. 2012. ILR: Where'd My Gadgets Go?. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, USA, 571–585.

[21] 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, NY, USA, 1470–1486.

[22] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 380–392.

[23] 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, NY, USA, 272–280.

[24] 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*. IEEE, San Jose, CA, USA, 339–348.

[25] 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 Association, Berkeley, CA, USA, 147–163.

[26] 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*.

[27] 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*. 280–291.

[28] 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, NY, USA, 190–200.

[29] Microsoft. [n. d.]. Data Execution Prevention. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention. Accessed Jan 24, 2020.

[30] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 1187–1198.

[31] 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, NY, USA, 245–258.

[32] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *ACM Sigplan Notices*, Vol. 45. ACM, New York, NY, USA, 31–40.

[33] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 89–100.

[34] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 1317–1328.

[35] 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, NY, USA, 914–926.

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

[37] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*. IEEE, San Jose, CA, USA, 49–58.

[38] Aleph One. [n. d.]. Smashing The Stack For Fun And Profit. http://phrack.org/issues/49/14.html. Accessed Jan 24, 2020.

[39] Tristan Ravitch. [n. d.]. Whole Program LLVM. https://github.com/travitch/whole-program-llvm. Accessed Jan 24, 2020.

[40] Gera Riq. [n. d.]. Advances in format string exploitation. http://phrack.org/issues/59/7.html. Accessed Jan 24, 2020.

[41] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 1–14.

[42] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. 552–561.

[43] 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, San Jose, CA, USA, 574–588.

[44] 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, NY, USA, 265–266.

[45] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. 2016. Eliminating Redundant Bounds Checks in Dynamic Buffer Overflow Detection Using Weakest Preconditions. *IEEE Trans. Reliability* 65, 4 (2016), 1682–1699.

[46] 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, San Jose, CA, USA, 48–62.

[47] The PaX Team. [n. d.]. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt. Accessed Jan 24, 2020.

[48] 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 Conference on Security Symposium*. USENIX Association, Berkeley, CA, USA, 941–955.

[49] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA, 157–168.

[50] 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 Conference on USENIX Security Symposium)*. 1805–1821.

[51] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2014. WPBOUND: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. In *Proceedings of the 25th International Symposium on Software Reliability Engineering*. 88–99.

[52] 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*.

[53] 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*.

[54] 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, DC, USA, 559–573.

[55] Rui Zhao, Chuan Yue, and Qing Yi. 2015. Automatic detection of information leakage vulnerabilities in browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*. 1384–1394.

[56] 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*. 292–302.