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.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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 $\frac{1}{2^{60}}$ 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.

![Figure 2: A buffer overflow vulnerability.](image)

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.

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].

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].

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 $M > N$, 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 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.

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

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.
We focus on protecting return addresses by adopting the same BarRA. Finally, the instrumented assembly code is assembled and linked lightweight runtime rerandomization on abstract return addresses. per, generates instrumented assembly code that is amenable to a single LLVM-IR (known as bitcode). We use a compiler wrap-piler tool chain, as shown in Figure 6. Given a C/C++ program, its 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 callees functions (Figure 4), BarRA instruments all the calls separately in order to also protect their return edges using CFI [1].
void read_and_echo(void) {
    char buf[BUFSIZE];
    gets(buf);
    printf(buf);
}

void ... + BAR_randval
...
2 + BAR_randval
...
ret addr ret id
(c) BARRA stack(b) Call stack

ret addr ret id

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_BAR_stack() 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 10, we have two return ids, 2 for read_and_echo() and 4 for gadget A. At the program startup t1, BAR_randval = 200 is generated. Just before read_and_echo() is called (lines 12-15 in

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.
BarRA stack

(a) Algorithm for runtime rerandomization

01 void * all_ret_addr[N]; // all return addresses
02 void * BAR_mtable[M]; // BarRA mapping table
03 int curDelta;

04 void update_BARA_stack(unsigned long rbp){
05 while(rbp != STACK_END){
06    unsigned long * pRetId = (unsigned long *) rbp;
07    (*pRetId) = (*pRetId + curDelta) % M;
08    rbp = *((unsigned long *) (rbp + sizeof(unsigned long)));
09 }
10}
11
echo

12 void rerandomize(){
13  random(time (0));
14  curDelta = random() % M;
15  BAR_randval = (BAR_randval + curDelta) % M;
16  unsigned long rbp = GET_RBP();
17  update_BARA_stack(rbp);
18}

BAR_randval

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

BarRA stack

(circular queue)

0
N-1
M-1

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

initial 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 2 204 204 8204 8204 8204
first ROP gadget id 4 204 204 8204 8204 8204

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 shadow stack mechanisms 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...
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 BarRandval) 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 BarRandval 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#LOC</th>
<th>#Text</th>
<th>#Funs</th>
<th>#Calls</th>
<th>Inst Overheads</th>
<th>Code Size Increases</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shadow Stack</td>
<td>BarRA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shadow Stack</td>
<td>BarRA</td>
</tr>
<tr>
<td>bzip2</td>
<td>8381</td>
<td>83271</td>
<td>100</td>
<td>425</td>
<td>0.77%</td>
<td>3.74%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.56%</td>
<td>18.07%</td>
</tr>
<tr>
<td>gcc</td>
<td>517621</td>
<td>4434399</td>
<td>5572</td>
<td>52763</td>
<td>3.16%</td>
<td>4.14%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.13%</td>
<td>24.42%</td>
</tr>
<tr>
<td>gosmk</td>
<td>197303</td>
<td>1613664</td>
<td>2679</td>
<td>9980</td>
<td>5.65%</td>
<td>9.30%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.80%</td>
<td>8.21%</td>
</tr>
<tr>
<td>h264ref</td>
<td>51752</td>
<td>692436</td>
<td>590</td>
<td>3543</td>
<td>1.93%</td>
<td>4.63%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.32%</td>
<td>11.57%</td>
</tr>
<tr>
<td>hmmer</td>
<td>35992</td>
<td>345292</td>
<td>538</td>
<td>4086</td>
<td>0.57%</td>
<td>0.38%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.65%</td>
<td>23.15%</td>
</tr>
<tr>
<td>lbbm</td>
<td>1155</td>
<td>23518</td>
<td>19</td>
<td>71</td>
<td>-0.37%</td>
<td>5.17%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.31%</td>
<td>13.75%</td>
</tr>
<tr>
<td>libquantum</td>
<td>4357</td>
<td>43429</td>
<td>115</td>
<td>556</td>
<td>0.54%</td>
<td>1.62%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8.82%</td>
<td>43.46%</td>
</tr>
<tr>
<td>mcf</td>
<td>2685</td>
<td>17381</td>
<td>24</td>
<td>79</td>
<td>4.35%</td>
<td>7.02%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.02%</td>
<td>7.02%</td>
</tr>
<tr>
<td>mile</td>
<td>15042</td>
<td>141659</td>
<td>235</td>
<td>1619</td>
<td>1.07%</td>
<td>1.47%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.64%</td>
<td>25.26%</td>
</tr>
<tr>
<td>perlbench</td>
<td>168274</td>
<td>1419350</td>
<td>1870</td>
<td>15334</td>
<td>3.34%</td>
<td>2.63%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.03%</td>
<td>17.64%</td>
</tr>
<tr>
<td>sjeng</td>
<td>13847</td>
<td>155749</td>
<td>144</td>
<td>1361</td>
<td>4.04%</td>
<td>5.57%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.56%</td>
<td>17.64%</td>
</tr>
<tr>
<td>sphinx</td>
<td>25104</td>
<td>205120</td>
<td>369</td>
<td>2733</td>
<td>0.20%</td>
<td>0.59%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.86%</td>
<td>28.78%</td>
</tr>
<tr>
<td>astar</td>
<td>5842</td>
<td>49896</td>
<td>153</td>
<td>665</td>
<td>3.93%</td>
<td>4.44%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7.52%</td>
<td>36.92%</td>
</tr>
<tr>
<td>dealII</td>
<td>198642</td>
<td>4165302</td>
<td>19234</td>
<td>93380</td>
<td>29.81%</td>
<td>20.26%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12.07%</td>
<td>57.62%</td>
</tr>
<tr>
<td>namd</td>
<td>3188</td>
<td>477495</td>
<td>140</td>
<td>1497</td>
<td>0.68%</td>
<td>0.54%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.72%</td>
<td>5.99%</td>
</tr>
<tr>
<td>omnetpp</td>
<td>48040</td>
<td>800247</td>
<td>2765</td>
<td>21041</td>
<td>15.69%</td>
<td>17.56%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9.08%</td>
<td>49.38%</td>
</tr>
<tr>
<td>povray</td>
<td>155163</td>
<td>1131443</td>
<td>2023</td>
<td>15114</td>
<td>17.14%</td>
<td>19.05%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.64%</td>
<td>28.53%</td>
</tr>
<tr>
<td>soplex</td>
<td>41428</td>
<td>442572</td>
<td>1542</td>
<td>9867</td>
<td>8.63%</td>
<td>9.30%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.80%</td>
<td>8.21%</td>
</tr>
<tr>
<td>Xalan</td>
<td>553631</td>
<td>5516159</td>
<td>29743</td>
<td>105849</td>
<td>1.12%</td>
<td>0.07%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13.31%</td>
<td>55.99%</td>
</tr>
<tr>
<td>Total</td>
<td>2,047,447</td>
<td>21,758,382</td>
<td>67,855</td>
<td>339,983</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>5.38%</td>
<td>6.09%</td>
<td>6.15%</td>
<td>29.44%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Read/Write Primitive</th>
<th>Format String</th>
<th>Example</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute write primitive</td>
<td>%ku%hn</td>
<td>printf(“%65535u%hn”, val, &amp;cnt)</td>
<td>Write two bytes, 0xFFFF (65535), to &amp;cnt</td>
</tr>
<tr>
<td>Absolute read primitive</td>
<td>%s</td>
<td>printf(“%s”, &amp;str)</td>
<td>Read the content at &amp;str</td>
</tr>
<tr>
<td>Relative read primitive</td>
<td>%08x</td>
<td>printf(“%08x%08x%08x%08x%08x”c)</td>
<td>Read the first five anonymous arguments</td>
</tr>
<tr>
<td>Relative read primitive</td>
<td>%k$u</td>
<td>printf(“%3$u”)</td>
<td>Read the 3rd anonymous argument</td>
</tr>
</tbody>
</table>

(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 dup2() 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 format 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 0x7f4740.

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
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, getu() 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 the 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 "%0x"s used in the malicious format string.

Let us focus on describing how 0x7e26da0 (the address of system()) is written into the shadow stack 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 "%0x" 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 16×3 + 8 = 50 characters after having processed lines 48-51. Next, printf() parses "%279760u" in line 52, causing the current anonymous argument on the call stack, i.e., 0x41414141 to be written as 27976 characters. On encountering the first "%hn" in line 52, printf() has printed 16×3 + 8×5 + 27976, i.e., 0x6060 characters, which are the lower two bytes of the address of the function system(), i.e., 0x7e26da0. 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., 0x7e26da0 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).

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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow Stack</td>
<td>Reducing Buffer Overflow Attacks</td>
<td>Vulnerable to Information Leakage Attacks</td>
</tr>
<tr>
<td>Control flow integrity (Backward)</td>
<td>Reducing Available ROP Gadgets</td>
<td>Still Permitting Illegal Return Addresses</td>
</tr>
<tr>
<td>Runtime Rerandomization (Concrete)</td>
<td>Alleviating Information Leakage Attacks</td>
<td>Problematic Pointer Tracking</td>
</tr>
<tr>
<td>BarRA (Abstract)</td>
<td>All the Above but without Pointer Tracking</td>
<td>Probabilistic Protection</td>
</tr>
</tbody>
</table>

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 0x0000000c, 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., $\frac{1}{M}$ 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 $M \gg N$. 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 $\frac{1}{M-N}$.

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).
©2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. ICSE ’20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, June 2020, Pages 258-270, https://doi.org/10.1145/3377811.3380439.

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 $\frac{N}{M} = \frac{1}{20}$.

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 $\frac{N}{M} = \frac{1}{20}$ (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, 487]. 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.


