ReRanz: A Light-Weight Virtual Machine to Mitigate Memory Disclosure Attacks

Zhe Wang[1,2], Chenggang Wu[1] *, Jianjun Li[1], Yuanming Lai[1],
Xiangyu Zhang[3], Wei-Chung Hsu[4], Yueqiang Cheng[5]

[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences,
[2]University of Chinese Academy of Sciences, [3]Dept. Computer Science, Purdue University,
[4]Dept. Computer Science & Information Engineering, National Taiwan University, [5]Acetti Software

[1]{wangzhe12, wucg, lijianjun, laiyuanming}@ict.ac.cn, [3]xyzhang@cs.purdue.edu, [4]hsuwc@csie.ntu.edu.tw,
[5]ycheng@acettisoftware.com

Abstract

Recent *code reuse* attacks are able to circumvent various *address space layout randomization* (ASLR) techniques by exploiting *memory disclosure* vulnerabilities. To mitigate sophisticated *code reuse* attacks, we proposed a light-weight virtual machine, ReRanz, which deployed a novel continuous binary code re-randomization to mitigate memory disclosure oriented attacks. In order to meet security and performance goals, costly code randomization operations were outsourced to a separate process, called the "*shuffling process*". The shuffling process continuously flushed the old code and replaced it with a fine-grained randomized code variant. ReRanz repeated the process each time an adversary might obtain the information and upload a payload. Our performance evaluation shows that ReRanz Virtual Machine incurs a very low performance overhead. The security evaluation shows that ReRanz successfully protect the Nginx web server against the Blind-ROP attack.

Categories and Subject Descriptors D.3.4 [*Software*]: Processors—Code generation; Translator writing systems and compiler generators; D.4.6 [*Operating Systems*]: Security and Protection

General Terms Design, Security

Keywords ReRanz, virtual machine, memory disclosure, re-randomization, shared memory

* To whom correspondence should be addressed.

VEE '17, April 08 - 09, 2017, Xi'an, China

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4948-2/17/04. . . $15.00

DOI: http://dx.doi.org/10.1145/3050748.3050752

1. Introduction

Memory corruption techniques [42] allow adversaries to hijack a program's control flow. This has created a war on computer security where defensive researchers mend loopholes in code as offensive researchers come up with new attacks. This war has been an arms race between offense and defense in the past decades.

Data execution prevention (DEP) protection mechanism was proposed to prevent *code injection* attacks [31]. However, the adversaries adapted quickly and evolved to *code reuse* attacks [12, 13, 20, 34], such as *Return-oriented Programming* (ROP) [34]. Instead of injecting code, they used the existing code fragments (called *gadget*s) residing in the victim application's address space. To mitigate *code reuse* attacks, *Address Space Layout Randomization* (ASLR) [43] had been proposed. It randomizes the base address of executable modules thereby making the gadgets' locations hard to guess. In order to further raise the bar of cracking the code layout, several fine-grained ASLR [8, 17, 24, 28, 29, 32, 44] techniques with high randomization entropy have also been introduced to enhance ASLR. Unfortunately, the adversaries invented new *memory disclosure* attacks [41] that leaked the target memory content directly on-the-fly and then performing *code reuse* attacks. To mitigate such attacks, *runtime code randomization* or *re-randomization* techniques [10, 14, 19, 30, 45], which randomize application code on-the-fly, have also been proposed recently.

Learning from the war, we note that although the new defense techniques cannot guarantee absolute security for a system, they could raise the difficulty for attackers. There are only a small number of defense techniques, such as DEP and ASLR, that are actually deployed in practice. Deployed techniques are cost-effective, can be applied to executables directly, and preserve the application binaries' common features such as *self-referencing code*[1], multi-threading and multi-processes. Recent *re-randomization* enhanced ASLR further raises the bar for attackers. Even if the attackers are aware of *memory disclosure* vulnerabilities, they can hardly launch *code reuse* attacks. However, these techniques suffered from serious deployment issues either because they require the source code to be available (so they are not applied to commercial or legacy applications) [10, 14, 19, 45], they incur unacceptable performance overhead (e.g., over 1000-fold) [30], or they compromised the binaries' common features (e.g., they cannot support *self-referencing code*). In this paper, we tried to address these three problems to ease the deployment of ASLR with runtime re-randomization. We present a novel, light-weight virtual machine, RERANZ, which continuously randomizes a binary executable together with dynamically loaded libraries. RERANZ not only performs re-randomization at fine granularity with low overhead, but also retains the binaries' common features.

RERANZ protects the subject process's code by setting its permission to *read-only* and executes the code through a Code Cache which is a software controlled memory region containing re-translated/randomized code. To randomize code efficiently and preserve the DEP protection mechanism, we devise a *shared memory* based shuffling mechanism. We set up a dedicated process, called the "*shuffling process*", to perform randomization. The Code Cache is shared between the main process and the shuffling process. The shuffling process generates several code variants periodically in its memory space. At any given time, there is only one code variant mapped to the shared Code Cache. When the protected process needs re-randomization, another code variant is mapped to the Code Cache, replacing the current one. Re-randomization is triggered when an *input* system call follows some *output* system calls since last randomization [10]. To reduce the overhead of I/O intensive applications (e.g., web servers), we also develop a trade-off method which could be a user controllable option. Because the code layout is changed continuously, re-randomization needs to update all code position related pointers when performing re-randomization. That is a prominent challenge and previous works did not solve it well. One key difference of RERANZ is to not mutate code pointer values in code variants. Instead, these pointers are dynamically translated to the randomized locations when they are used to change control flow. More importantly, RERANZ is carefully designed so that the translation cannot be reused by the adversaries.

In summary, the work makes the following contributions:

- We proposed a new light-weight virtual machine, RERANZ, to mitigate *memory disclosure* attacks. It has been successfully tested on commercial applications.

[1] Self-referencing code usually treat the code pointers as data pointers and use these data pointers to read the content. For example, libunwind library [4] uses the return address to read its own code and checks whether the instructions are PLT encoded.

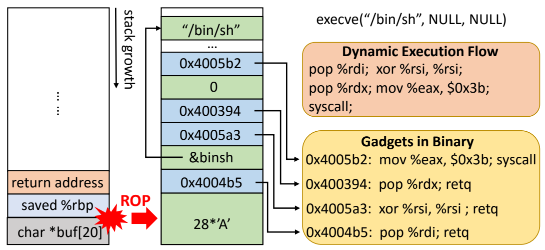


Figure 1: Return-Oriented Programming to start a shell.

- Using an innovative shuffling process to generate and hide code variants in its memory space in parallel and modify the protected process's page table to switch the new code variants.
- Re-randomization is achieved without requiring mutating code pointers, which is critical for the practicality of the technique. While this is enabled by dynamic address translation, the translation itself may introduce security risks, RERANZ invents new methods to lower such risks.
- An improved feedback based re-randomization strategy is proposed and implemented. Compared with the TASR's strategy [10], the new method achieves much better performance on I/O intensive applications.
- The RERANZ virtual machine is evaluated on a Linux/X86_64 platform. Our evaluation shows that RERANZ retains the full functionalities of the original application. The performance evaluation shows that RERANZ incurs 9.7% overhead for the multi-threaded Parsec-2.1 [9] benchmark, 6% for SPEC CPU2006 [23], and 10% for Apache/Nginx web servers [1, 5] running in the batch mode. The security experiments show that the prototype prevents the Blind ROP attack [11] successfully.

The rest of the paper is organized as follows. Section 2 reviews the related work and gives our motivation. Section 3 explains the threat model under which RERANZ operates. Section 4 presents the design and implementation of RERANZ and Section 5 describes some practical challenges and some optimizations. Section 6 provides the security and performance evaluation and Section 7 discusses RERANZ's security design and our planned future work.

2. Background and Related Work

2.1 Return-Oriented Programming (ROP)

ROP attack [34] is used to hijack the intended execution flow of an application and performs malicious operations without injecting any new code. Figure 1 shows an ROP attack that spawns a command shell. It uses a stack buffer overflow vulnerability to write data (called *payload*) beyond the bound of the buffer and then overwrite the critical control-flow information (e.g., *return address*). The payload consists of pointers, where each pointer refers to a small code piece (called *gadget*). Each gadget consists of several instructions that ends with a *ret* instruction. Once a gadget has executed and the top of the stack points to the address of the next gadget, the exploit continues. After several years' development, several evasive variants of ROP had been invented. They could use diverse gadgets (e.g., ending with *indirect jump*s and *indirect call*s) to build up the payload [12, 13, 20].

2.2 Load-time Address Space Layout Randomization

Load-time ASLR had been proposed to mitigate *code reuse* attacks [43] by making the addresses of gadgets unpredictable at *load-time*. Module-level ASLR is widely deployed. But adversaries could also use several leaked pointers (e.g., the base address of libraries) to relocate the gadgets' address in the payload. So several fine-grained randomization techniques further enhanced ASLR by permuting the location of functions [28] and randomizing the location of code pages [8], basic blocks [17, 29, 32, 44] or instructions [24]. It is difficult for adversaries to relocate (through several leaked addresses) or guess (through *brute force* attacks [35, 39]) the gadgets' location under these techniques.

2.3 Memory Disclosure Attacks

Although *load-time* randomization techniques ensure that the code layout will be different every time a process is running, the code layout will remain unchanged throughout the execution. Therefore, new *memory disclosure* attacks [41] appeared, such as JIT-ROP [40], which can circumvent these protections without requiring precise knowledge of the code layout. Instead, it can use any code pointers (e.g., *return addresses*) to implement the attack. Based on the leaked pointers, JIT-ROP discloses the memory contents by recursively searching for code pointers and then uses the discovered code pages to generate the ROP payload on-the-fly. JIT-ROP gathers the memory pages without crashing the victim application.

The threat of memory disclosure is not limited to JIT-ROP. Another popular attack is the *clone-probing* attack, such as Blind ROP [11], which focus on daemon web servers. A daemon web server consists of a daemon process and multiple worker processes forked by the daemon process. If a worker process crashes, a new worker process will be forked by the daemon process. Since the memory layout of worker processes is the same as the daemon process', adversaries can repeatedly probe the worker processes and then analyze the characteristics of the corresponding responses (e.g., *crash*, *block*, *closed* or *stays open*) to remotely obtain information of the memory layout.

2.4 Previous Efforts of Re-randomization

Re-randomization, which randomizes the application code on-the-fly, is not a new idea. Many researchers had noted that *re-randomization* can mitigate *memory disclosure* attacks. If code is re-randomized between the time that it is leaked and the time a *payload* is invoked, the attack fails because the *gadget*s do not longer exist.

There are two main challenges to actually deploy effective re-randomization. (a) All code location related pointers need to be updated during code re-randomization. Hence, precisely identifying and tracking such pointers during execution is a prominent problem. (b) At what time and how frequent should re-randomization be performed is crucial to both efficiency and security.

To address the first challenge, all recent re-randomization techniques require the compiler's assistance. OS-level ASR [19] uses the LLVM [3] to instruments the IR to track the code pointers. TASR [10] modifies the GCC compiler to collect the location information of code pointers. Remix [14], RuntimeASLR [30] and Shuffler [45] need to recompile the source code with special options to obtain the initial code pointers (e.g., Remix and Shuffler need '-Wl, -g, -gdwarf2, -fno-omit-frame-pointer, -z initfirst' options to provide symbols and Dwarf unwind information. RuntimeASLR needs the '-PIE' or '-fPIC' options). Specially, Shuffler replaced the code pointers with the indices of a global table of code pointers. When performing re-randomization, it only relocated the global table. Because the semantics of code pointers have been changed, the *self-referencing code* cannot be supported. Besides Shuffler, Remix and OS-level ASR also cannot support *self-referencing code* due to its fine-grained re-randomization (destroying the original code layout deeply). Although RuntimeASLR needs few changes in the compile options, it utilizes dynamic binary instruction to track code pointers that incurs very high overhead (over 1000-fold of application execution time).

To address the second challenge, OS-level ASR, Remix and Shuffler choose to perform re-randomization at a fixed time interval. Because the adversaries could complete the attack between the adjacent randomizations, this randomization strategy is not secure enough. RuntimeASLR only performs the re-randomization when the parent process forks a child process. So it cannot prevent the JIT-ROP attack. TASR proposed a different re-randomization strategy: (a) re-randomize before any *input* system call that following one or more *output* system calls; (b) re-randomize before *fork* and *vfork* system calls. In TASR, any *output* system call may be used to leak the information (i.e., pointers and code content) and any *input* system call may be used to upload a payload. So strategy (a) re-randomizes before the adversary uploads a payload to invalidate the gadget addresses in the payload. Strategy (b) is used to prevent *clone-probing* attacks. However, based on this re-randomization strategy, TASR could cause prohibitively high overhead for I/O intensive applications due to excessive re-randomization.

Re-randomization needs to re-randomize all codes in the victim applications' memory space. Besides Shuffler and Remix, most of re-randomization techniques could do that. Shuffler did not re-randomize the loader library and Remix only re-randomized part of functions that had enough

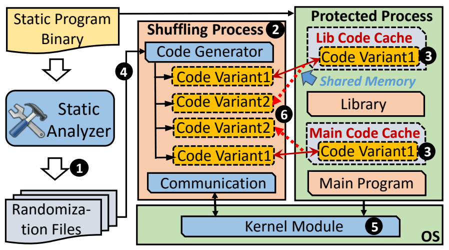


Figure 2: Architecture overview of ReRanz.

paddings. So the adversaries may have chance to use these un-randomized codes to conduct *memory disclosure* attacks.

2.5 Motivation for ReRanz

Recent re-randomization techniques have not been deployed in practice due to four issues: (a) They may require source code to be available so that they could not be applied to commercial or legacy applications; (b) They involve high performance overhead; (c) They do not preserve common application binary features, such as *self-referencing code*; and (d) They do not re-randomize all codes. In this paper, ReRanz is specifically targeted at addressing these issues to make the deployment of re-randomization practical.

3. Threat Model and Assumptions

In our threat model, the adversary has the following two abilities. (a) Read and write arbitrary memory locations in the victim process; (b) Ability to crash the worker process of a daemon repeatedly, but not crash a victim process without *self-rebooting* functionality. ReRanz focuses on preventing *code reuse* attacks from adversaries with the above capabilities. Attacks that exploit data layout [25, 37] to leak information implicitly and *side-channel* attacks [22, 26, 27, 38] are out of our scope. We also assume that the target system is equipped with the following protections which are very popular in existing commercial systems.

No Executable-and-Writable Memory All memory pages are either marked as *executable* or *writable*, thus preventing *code injection* attacks.

Load-time ASLR *Load-time* ASLR is the first step to shuffle the layout of a victim process. In this work, we only randomize code and data segments that are supposed to be protected by the *load-time* randomization.

4. Design and Implementation

4.1 System Overview

The overall architecture of ReRanz is depicted in Fig. 2. To show ReRanz is practical on real executables, we develop and deploy our system on the Linux/X86_64 platform. ReRanz begins by analyzing the executable and its libraries, and then generate a randomization file for each module in an **Offline Static Analysis Phase**. This phase is used to accelerate randomization at runtime. It is optional and can be

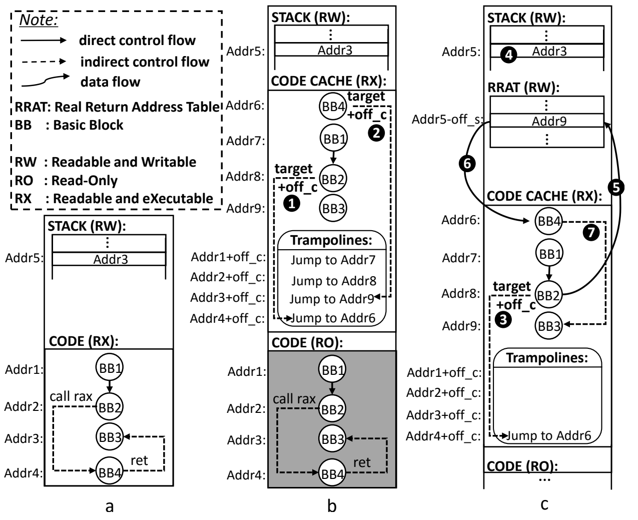


Figure 3: Three memory layouts of the victim process with different protection strategies. (a) shows the victim process without protection. (b) shows the basic method and (c) shows the enhanced strategy.

integrated into the second phase. It disassembles the executable, partitions it into basic blocks and then transforms the instructions in each basic block (❶). The result of randomized blocks is stored in a randomization file.

The task of the **Load-time Initialization Phase** is to allocate the Code Cache, and generate the first code variant into it. We outsource the costly re-randomization operation to a separate user space process, called the "*shuffling process*", which launches on a different core (❷). This reduces the impact of the latency introduced by the randomization work. ReRanz uses a kernel module to monitor the initialization of the protected process's memory layout and allocate the Code Cache for each module. Meanwhile, the original code region is set with *read-only* permission. The Code Cache of the protected process is shared with the shuffling process so that it can generate the code variant in the shared space directly (❸). The code variant is generated based on the pre-generated randomization files of all modules (❹) in the first phase. It is randomized at the basic block granularity (i.e., basic block reordered). Based on the *shared memory* mechanism, the Code Cache is given the *readable* and *executable* permissions in the protected process, whereas it has the *readable* and *writable* permissions in the shuffling process. Because the Code Cache does not have *writable* permission for the protected process, ReRanz could preserve DEP principle throughout the entire execution.

In the **Runtime Re-randomization Phase**, the kernel module monitors the normal execution of the protected process and the shuffling process generates new code variants in parallel. Once a specific pattern of system calls is detected (❺), the kernel module flushes the old code variants and replace with a new one in the Code Cache (❻). This is done by modifying the page table of the protected process to map the

Instruction	Type	Description	Design
Indirect Call	**C-Type**	Jump to function entries (e.g., virtual function calls and function pointer arrays).	Trampoline
Return	**R-Regular**	Return address is pushed on the stack by call instructions.	RRAT
	R-Signal	Return address is pushed on the signal stack by OS.	RRAT
	R-Exception	Return address is updating dynamically (i.e., *_Unwind_RaiseException()* in C++ exception).	Catch Table
Indirect Jump	**J-Table**	Jump to use the jump table (e.g., *swith/case* statement).	New Table
	J-Function	Jump to function entry (e.g., *_dl_runtime_resolve*).	Trampoline
	J-Middle	Jump to the middle of the current function (**special behavior**).	Trampoline
	J-Exception	Jump to the middle of other functions (i.e., *longjmp/siglongjmp*).	Setjmp Table

Table 1: A summary of indirect branch instructions.

Code Cache with a new code variant from the shuffling process. By leveraging the isolation of processes, the new code variants are hidden in the protected process's virtual memory space. It is important that the adversaries cannot leak and use them before they are switched in.

4.2 Two Key Design Decisions

4.2.1 Code Randomization without Tracking Pointers

Both explicit and implicit pointers to code exist in programs. When shuffling codes, we must guarantee such pointers will correctly point to their new locations. However, identifying such pointers is a challenge, especially when the source code is not available.

An important design decision is to avoid identifying and updating code position related pointers. Instead, ReRanz dynamically redirects the pointer (in the original code space) to its respective address in the code variant when it is de-referenced. All code pointers in a randomized code variant remain intact by pointing to the locations in the original code, they are merely redirected to randomized new locations. This property is required for *self-referencing code*.

ReRanz instruments all indirect branch instructions (i.e., *ret*, *indirect call* and *indirect jump* instructions) to perform runtime address translation. Upon each re-randomization, ReRanz only needs to update the address translation table. To achieve efficient translation, ReRanz maintains a set of trampolines in the Code Cache. As shown in Fig. 3(b), each basic block has a corresponding trampoline located at a place with a fixed offset *off_c* (e.g., translations ❶❷) from the original basic block and the jump target of a trampoline is the randomized location. When re-randomization occurs, we only need to reorder the basic blocks in the Code Cache and update all the trampolines and the current program counter.

Security Flaw of the Basic Design To launch an attack, the adversaries could chain basic blocks (named *useful gadgets*), which have the corresponding trampoline, and ended with an indirect branch instruction de-referencing the code pointer to trampolines. Then they can use the *useful gadget* addresses pointing to the original code area in the payload. With the help of the trampolines, the gadget addresses in the payload can be translated to the randomized locations. If this imaginary attack turns out to be true, ReRanz fails to protect the program. A recent randomization technique, *instruction displacement*[2] [29], had also noted this problem that the adversaries could utilize the "basic block entry" gadgets (similar to our *useful gadgets*) to conduct *code reuse* attacks. However, that work did not adopt any protection on these gadgets, it only showed that these gadgets were infrequent.

Enhanced Design The root cause of the above vulnerability is that the adversary might exploit the process of the dynamic code translation. We attempt to minimize such risks through two ways: (a) Reduce the pointer de-reference operations of accessing trampolines. (b) Reduce the number of trampolines. Both approaches leave very few blocks as gadget candidates for adversaries to exploit.

For aspect (a), we eliminate the use of trampolines for all *ret*s and a subset of *indirect jump*s. To understand the behavior of indirect branch instructions, we analyzed 166[3] Linux binaries semi-automatically and classified them in Table 1. *Ret* can be classified into three types: **R-Regular**, **R-Signal** and **R-Exception**. To eliminate the use of trampolines for **R-Regular**, we leverage a *Real Return Address Table* (RRAT) to store the real return addresses during the execution of *call* instructions and *ret* is transformed to load the jump target from RRAT (as shown in Fig. 3(c)❸-❼). We also utilize the RRAT for **R-Signal**, but it is somewhat different in that the return address of signal handler is pushed by the OS during delivering a signal. So the kernel module is responsible for storing the real return address in RRAT. The return address of **R-Exception** is updated at runtime and this mostly occurs in exception handling of C++ programs (i.e., *try/catch* statements). When the *try* statement throws an exception, *_Unwind_RaiseException()* of *libgcc_s.so* will find the corresponding *catch* statement and modify its own return address to point to the location of the *catch* statement. So for **R-Exception**, *ret* is transformed to search an *catch table*. The table stores the mapping of all *catch* statements from

[2] It moves the original basic block in a random location and erases the original code. To maintain the original semantics of the code, the randomized basic blocks are linked with the rest of the code by using direct jump instructions which are placed at the original locations.

[3] Including SPEC (29), Parsec (13), Web servers (3), Browsers (3), Libraries (19) and Shell commands (33 for text operations, 24 for file operations and 42 for others)

Method	Description	Reliability
FE-M1	Function entries recorded in (dynamic) *symbol tables*.	High
FE-M2	Function entries recorded in *relocation tables* (e.g., global function).	High
FE-M3	Function entries are jump targets of direct *call* instructions.	High
FE-M4	Potential function prologs (e.g., decreasing *%rsp* and saving *callee-saved* registers).	High
FE-M5	Function entries are jump targets of the direct tail call instruction. This is often occurred in the optimization of compilers. The direct/indirect *call* instruction is replaced with a direct/indirect *jmp* instruction to reuse the caller's stack frame. The *jmp* instruction is often preceded with potential function epilogs (e.g., increasing *%rsp* and restoring *callee-saved* registers) and some argument registers' assignment.	High
FE-M6	Function entries are preceded with paddings (usually filled by *00H* or *'nop'* instruction in ELF). Most compilers often insert paddings to align functions and loops to a power-of-two (often 16-byte aligned) boundary in the cache.	Middle

Table 2: Methods to recognize function entries.

their original locations (extracted from *.gcc_except_table* in binaries) to the new randomized locations. *Indirect jump*s can also be classified into four types: **J-Table**, **J-Function**, **J-Middle** and **J-Exception**. We only eliminate the use of trampoline for **J-Table** and **J-Exception**. For **J-Table**, the *indirect jump*s are transformed to use a *new jump table* that contains all the translated addresses. The table is derived from the original jump table by translating each value to new randomized locations. The jump tables in the shared library usually record offset instead of absolute address. For these tables, the new tables contain the relocated offsets. For **J-Exception**, the *indirect jump*s are transformed to look up a *setjmp table* that stores the mapping of all original return addresses at the callsites of *setjmp()/sigsetjmp()* to the new locations. Because the tables used for **R-Exception**, **J-Table** and **J-Exception** are constant for a code variant, so they can be generated together with the variant into the Code Cache.

After the aforementioned transformation, the trampolines are only used for the jump target basic blocks of **C-Type**, **J-Function** and **J-Middle**. So for aspect (b), we need to reduce the corresponding trampolines of other basic blocks. From Table 1, we can see that the jump targets of **C-Type** and **J-Function** are all function entries. We also observed that **J-Middle**'s targets are not the target of any direct branch instructions. So we design a conservative method that follows control flow edges (produced by direct branch instructions and **J-Table**) from possible function entry basic blocks (marked as requiring trampolines) to recognize intra-procedural basic blocks (marked as not) and the exploration is bounded by basic blocks that have been analyzed before. The remaining basic blocks are all marked as requiring trampolines. Because functions are difficult to recognize in binaries, we recognize function entries through methods in Table 2. The recognized function entries through **FE-M**(**1**-**5**) are more common. Because the paddings are inserted for functions and loops, **FE-M6** may mistakenly recognize the end of a loop as a function entry. So the final classification method is: **Step 1**, recognizing function entry basic blocks through **FE-M**(**1-5**) and then following control flow edges to classify basic blocks; **Step 2**, recognizing function entries again in un-reachable basic blocks through **FE-M6** and then following control flow edges to classify basic blocks; **Step 3**, marking the remaining basic blocks as requiring trampolines.

Proof-of-Concept The conservative classification ensures correctness (the remaining trampolines are enough for **C-Type**, **J-Function** and **J-Middle**) for two reasons: (1) Because **FE-M5** recognizes direct tail calls, the function entries will not exist in control flow edges. Even though **FE-M**(**1-6**) may not recognize all function entries, the remaining function entries will be handled conservatively at **Step 3**; (2) Based on our observation of **J-Middle**, the jump target basic blocks will not exist in control flow edges. So its targets will be marked as requiring trampolines at **Step 3**. Even if we recognize its target as a function entry by mistake, its targets will still be marked as requiring trampolines. But this method may recognize additional basic blocks as requiring trampolines, which exposes opportunities for adversaries to exploit. However, such extra basic blocks are rare, and our security evaluation shows that all the remaining *useful gadget*s are very difficult to launch a meaningful *code reuse* attack due to the lacking of the key types and having strong side effects (shown in Section 6).

Because most trampolines are eliminated, there are many holes left in the region of trampolines. To reduce the size of the code variant, the randomized basic blocks and trampolines are interleaved instead of separated actually.

4.2.2 Time to Perform Re-randomization

The most effective re-randomization strategy so far is TASR [10]. However, TSAR may cause prohibitively high overhead for I/O intensive applications due to frequent re-randomization. Our study shows that leaked information through *output* system calls are code pointers and code contents. Code pointers can be used to relocate the gadget addresses in payload or leak the code pages. Because the code is randomized with fine granularity in ReRanz, the adversaries could not use these pointers to relocate the payload (similar to *load-time* fine-grained randomization). So we only concerned about the leakage of code contents.

Original Basic Block: 0xbf1d

```
bf1d:  48 8b 88 48 04 00      mov %rcx, $0x448(%rax)
bf24:  48 3b 4a  f0           cmp %rcx, -0x10(%rdx)
bf28:  77 39                  ja 0xbf63
```

RelocatableBB: 0xbf1d

Encode

```
00:  48 8b 88 48 04 00   mov %rcx, $0x448(%rax)
07:  48 3b 4a f0         cmp %rcx, -0x10(%rdx)
0b:  0f 87 00 00 00 00   ja 0x11
11:  e9 00 00 00 00      jmp 0x16
```

RelaTable

No	*Type*	*RelaPos*	*Size*	*Addend*	*Value*
0	BRANCH	0x0d	4	-0x11	0xbf63
1	BRANCH	0x12	4	-0x16	0xbf2a

Figure 4: A basic block and its RelocatableBB.

To address the performance problem, we consider a trade-off that monitors the amount of data transmitted by *output* system calls to reduce the frequency of re-randomization. The intuition is that the sequence of *output* system calls before an *input* system call may not transmit a large amount of data. The amount of leaked code is hence insufficient to launch *code reuse* attacks. ROPecker [15] had observed that the adversary has a very low possibility to launch a meaningful *code reuse* attack if the size of executable code is smaller than 20KB. Inspired by this observation, we monitor the amount of transmitted data by *output* system calls and re-randomize when the accumulated amount exceeds a certain threshold. Because recognizing the code contents in the transmitted data on-the-fly is a time consuming job, we conservatively treat all transmitted data as potential code contents. Our re-randomization strategy is described as follows: (1) re-randomization happens upon executing an *input* system call and the accumulated amount of leaked data (by the preceding *output* system calls) has exceeded a certain threshold; (2) re-randomize before any *fork*/*vfork*/*clone*[4] system call. Note that the amount of leaked data is reset every time re-randomization is performed. With a larger threshold, we obtain lower security but with lower overhead. So it is a trade-off for users to consider.

4.3 Offline Static Analysis Phase

In this section and the following sections, we will discuss the individual phases in the workflow of ReRanz.

In the first phase, a static analyser searches for the libraries required by a given program and then analyzes the individual modules through the following five steps: (1) Disassemble the binary; (2) Extract some tables as in Table 1; (3) Partition the code into basic blocks; (4) Identify the basic blocks that require trampolines as we discussed in Section 4.2.1; (5) Transform instructions for randomization. The first three steps are similar to existing works [32, 44]. Step 4 has been described in the preceeding subsections. So we shall focus on the last step here.

[4] Besides fork and vfork, the program could also use the clone system call with 'CLONE_VM' flag to start a new process that is missed in TASR.

Original Code	Transformed Code
400564: mov **0x200a8d**(%rip),%rax 40056b: …	4000f842: mov **-0x3fa0e851**(%rip),%rax 4000f849: …
4008ea: lea %rbp, **0x200527**(%rip) 4008f1: …	40000064: lea %rbp, **-0x3f9ff253**(%rip) 4000006b: …

Note: 0x40056b + 0x200a8d = 0x4000f849 - 0x3fa0e851 = 0x600ff8
0x200527 + 0x4008f1 = 0x4000006b - 0x3f9ff253 = 0x600e18

Figure 5: Example of Position-Independent Code.

4.3.1 Generate Relocatable Basic Block

For each basic block, ReRanz generates a RelocatableBB that contains a modified basic block followed by a relocation table that records the addresses that need to be replaced during randomization. After randomization, only the modified basic block is copied to the Code Cache and relocated based on the relocation table. Figure 4 shows an example of RelocatableBB. In the original basic block, the address '*0xbf1d*' represents the offset relative to the header of the binary. A *direct jump* is added in the RelocatableBB to link up with the *fall-through* basic block. The relocation table contains multiple entries, each for an address in the basic block that requires relocation during randomization. Each entry contains the type (Column "Type" in Fig. 4), the relative position of the address in modified basic block (Column "RelaPos"), the size of the address (Column "Size"), and the information to compute the new value (Column "Addend" and "Value"). ReRanz has eight relocation types related to the direct branch instruction, position-independent code (PIC), RRAT, trampoline, return address and three new tables in Table 1. The meaning of 'Addend' and 'Value' depends on the relocation type. Due to the space limitation, we only introduce the 'BRANCH' type in Fig. 4: the relocated jump offset of '*ja*' = the position of RelocatableBB '*0xbf63*' − the position of RelocatableBB '*0xbf1d*' − '*0x11*'.

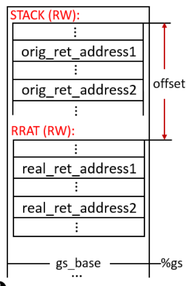
We also illustrate the transformation of PIC and instructions that related to RRAT and trampolines.

Position-Independent Code (PIC) There are two different implementations: (1) using "*callnext*; *pop %reg*" pattern to obtain the current address (mostly occurred on IA32 platform); (2) using the register *%rip* directly (only occurred on X64 platform shown in Fig. 5). Since ReRanz does not shuffle data locations and preserves the original code area, it only needs to ensure the transformed instruction yields the same data/code address as the original. Because the stack stores the original return address, we do not need to handle the first implementation. To handle the second case efficiently, ReRanz should make sure that the offset between the original code and the code variant is within ±31bits so that ReRanz could put the displacement of the memory operand in the relocation table and then relocate it during randomization.

Indirect Jumps and Calls **J-Function** and **J-Middle** are transformed to use trampolines. As shown in Fig 6(a), the *indirect jump* is redirected to the corresponding trampoline by adding a fixed offset. Because the '*add*' instruction will

Original	Transformed Code
jmp r/[m]	1: pushq r/[m]
	2: addq (%rsp), **$offset** #trampoline offset
	3: retq
call rel32	1: movl %gs:(**$addend** - 4)(%rsp), **$real_ret_l32**
	2: movl %gs:(**$addend** - 8)(%rsp), **$real_ret_h32**
	3: pushq **$orig_ret_l32**
	4: movl 4(%rsp), **$orig_ret_h32**
	5: jmp **target_addr**
call rel32 [OPT]	1: movq %gs:(**$addend** - 8)(%rsp), **$real_ret_l32**
	2: pushq **$orig_ret_l32**
	3: jmp **target_addr**
ret	1: addq %rsp, 8
	2: jmpq %gs: (**$addend** - 8)(%rsp)

Note:
The **bold** operands will have relocation entries in RelocatableBB.
a



➊ ***Instr. operand to access RRAT:***
Entry = -$offset(%rsp)

➋ ***Segmentation to Hide RRAT:***
Entry = %gs:$addend(%rsp)
$addend= -($gs_base+$offset)
b

Figure 6: Example of Position-Independent Code.

change the *%eflag* register, the control flow may be changed if the following instructions depend on *%eflag*. However, there is no instruction that uses *%eflag* prior to defining it within a function, so it is safe to change *%eflag* if the jump target is a function entry. When the jump target is not a function entry (handle the *indirect jump*s that not located in PLT actually), ReRanz uses '*pushf*' and '*popf*' instructions to protect *%eflag* before and after the '*add*' instruction. Besides updating the RRAT, *indirect call*s are similarly transformed to use trampolines. So detailed discussion is elided.

Direct Calls and Returns These instructions are handled by RRAT. To accelerate the access to RRAT, RRAT is designed to be indexed by a linear transfer function on the *%rsp* register. As shown in Fig. 6(b), RRAT is placed at a fixed offset from the original stack. If the top of stack is a return address, we can use the memory operand *-$offset(%rsp)* to access the corresponding real return address in RRAT (➊).To prevent leaking of the RRAT's location, we leverage the *%gs* segment register to hide RRAT. The *%gs* register points to an entry in the Global Descriptor Table (GDT) which contains the real address. In particular, we use *%gs:$addend(%rsp)* to access RRAT (➋).The kernel module is responsible for setting *%gs* at the start-up time of the protected process. Although there is no pointers that point to the RRAT, it is possible that the adversaries might use recent techniques [18, 21] to probe the locations of the RRAT linearly without crashing the program. In order to prevent from such probing, we placed two guard pages (before and after the RRAT) here to prevent from probing RRAT's location. As shown in Fig. 6(a), a *direct call* is transformed to five instructions (before any optimization). The first two instructions store the real return address and the following two instructions push the original return address on the stack. We could reduce the number of instructions, if the positions of the original code area and the Code Cache are both lower than 4G. *Ret* is transformed to use the real target in RRAT.

4.4 Load-time Initialization Phase

We design a kernel module to monitor the start-up of the protected process and allocate Code Cache for each module.

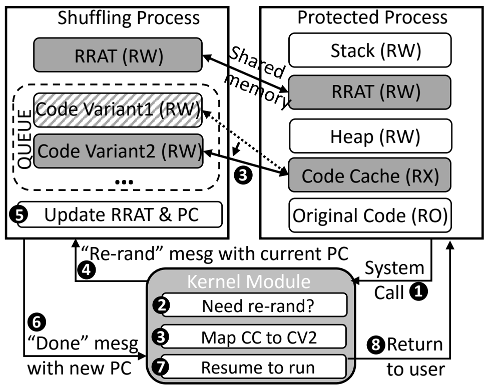


Figure 7: The runtime re-randomization process. The CC is short for the Code Cache, the CV is short for the Code Variant and the PC is short for the Program Counter.

To achieve this, the kernel module modifies the *system call table* to intercept a set of system calls.

Kernel Module Monitoring The kernel module intercepts the *execve* system call to monitor the start-up of a protected process. When the protected process is started, the kernel module allocates RRAT for the main stack and two Code Caches for the main program and *ld-linux.so*. Meanwhile, the kernel module starts a corresponding shuffling process and shares these two Code Caches and RRAT. The kernel module also monitors the process's library loading by intercepting the *mmap* system call. It allocates and shares the Code Cache for each module. In this phase, once the kernel module detects a new module, it will protect the original code region and send an '*Init*' message[5] to the shuffling process to generate this module's variant into the corresponding Code Cache. The entry of protected process is relocated to the randomized location and the following execution continues in Code Cache naturally.

Shuffling Process Initialization When the shuffling process starts, it establishes the socket connection with the kernel module immediately. Once it receives an '*Init*' message, it starts a child thread to generate N code variants for the corresponding module. The workflow of generating a code variant is the following. (1) Read the corresponding randomization files; (2) Place the trampolines; (3) Place all modified basic blocks in RelocatableBBs, new jump tables, setjmp table and catch table at random locations; (4) Relocate the basic blocks, trampolines and tables.

4.5 Runtime Re-randomization Phase

This phase performs re-randomization routinely based on our triggering policy (mentioned in section 4.2.2). As mentioned before, the kernel module intercepts three *input-like* system calls (i.e., *fork*, *vfork* and *clone*) and a set of *input* system call (i.e., *read*, *pread64*, *readv*, *recvfrom*, *recvmsg*, *preadv* and *mq_timedreceive*) and *output* system call

[5] Using Netlink socket to communicate between kernel and user spaces.

(i.e., *write*, *pwrite64*, *preadv*, *sendto*, *sendmsg*, *pwritev* and *mq_timedsend*). Each time the protected process invokes any of these system calls (❶ in Fig. 7), the kernel module determines if re-randomization shall be performed (❷). If so, the kernel module modifies the protected process's page table so that its Code Cache is mapped to a new code variant from the shuffling process (❸). And then the kernel module sends a '*Re-rand*' message to the shuffling process with the current program counter (❹). Once the shuffling process receives this message, it updates the program counter and the RRAT and then discards the old code variant (❺). After finishing these operations, the shuffling process sends a '*Done*' message to the kernel module with the new program counter (❻). When the kernel module receives this message, it finishes executing the current system call and resumes the execution at the new location (❼❽).

5. Challenges and Optimization

Multi-threading Since threads share the same address space but have their own private stacks, threads use the same code variant but have their own RRATs in ReRanz. Therefore, we only launch one shuffling process for all threads. The kernel module intercepts the *mmap* system call to capture the creation of a thread's stack[6] and then allocates the respective RRAT. The kernel module tracks I/O system calls among all threads. If an *input* system call of a thread triggers re-randomization, ReRanz pauses all threads, maps a new code variant and updates all RRATs and program counters.

Multiple Processes Multi-processed programs usually leverage shared memory regions to transmit data between processes. If the adversary uploads a payload to a process's shared memory region, the payload may be executed by other processes. So it is necessary to re-randomize all processes when any of the processes need to perform re-randomization. So the handling is similar to multi-threading and all processes share the same code variant to reduce the management difficulty of code variants. The kernel module tracks the I/O system calls for the entire process group. In fact, the kernel module monitors all processes that inherit the initial memory layout instead of the process group. This is because a child process may use system calls (e.g., *setsid*) to switch to another process group.

Linux Signals In Linux, if a process has previously registered a signal handler, the handler routine will be executed when the signal is thrown. In ReRanz, the location of the registered handler is not in the executable Code Cache. To address this problem, the kernel module intercepts the *sigaction* and *signal* system calls to register the location of randomized handler. In addition, the kernel module is also responsible for storing the randomized location of the restorer routine (i.e., *sigreturn*) in RRAT (mentioned in section 4.2.1). During re-randomization, the location information of handler and restorer recorded in kernel will be updated. The signal stack exists on the current stack by default. Hence the *call/ret* instructions in the handler can use the corresponding RRAT. However, Linux also allows a process to use its own signal stack by using the *sigaltstack* system call. To ensure the correctness of *call/ret* instructions in randomized handlers, the kernel module allocates a RRAT for each signal stack by intercepting the *sigaltstack* system call.

[6] It is allocated by the mmap system call with 'MAP_STACK' flag.

Dynamic Library Loading At runtime, the program may load and unload libraries. Hence dynamic loaded libraries should also be protected by ReRanz. The kernel module intercepts the *mmap* system call to monitor dynamic library loading of the protected process at runtime. When the protected process loads a library at runtime, the kernel module cancels its *executable* permission and allocates a Code Cache that is shared with the shuffling process. And then it sends a message to the shuffling process so that randomized code can be generated in the Code Cache. The kernel module also intercepts the *munmap* system call to monitor the operation of library unloading. If the protected process unloads a library, the kernel module will free its Code Cache and notify the shuffling process.

Destroying Caller-saved Registers As mentioned before, when transforming *indirect call*s and some *indirect jump*s, ReRanz pushes the jump targets of *indirect call/jump* on the stack and calculates the target address of the trampoline. These operations use several memory access instructions that may cause substantial slowdown. To reduce the number of memory accesses, we leverage *caller-saved* registers, which are saved before a function invocation by the caller and recovered after returning from the callee. It is hence safe to reuse *caller-saved* registers at the function invocation (only *indirect jump*s in PLT are handled actually).

Merging Fall-through Basic Block When generating RelocatableBBs, ReRanz generates a *direct jump* at the end of each basic block with a conditional jump (shown in Fig. 4). This *direct jump* is used to link up with the *fall-through* basic block and may incur high performance overhead due to degraded cache performance (In our experiment of SPEC CPU2006 benchmarks, these *direct jumps* will incur 176% performance overhead on average). We eliminate this instruction by merging the *fall-through* basic block with the current basic block. The randomization granularity is moved from basic block level to extended basic block level.

Reordering Extended Basic Blocks in Group A program's execution often exhibits both spatial and temporal code locality. However, the fine-grained randomization in ReRanz destroys the spatial locality and may seriously degrade the performance. To balance code locality and entropy, we reorder extended basic blocks in its group. When generating code variants, the shuffling process first sorts all extended basic blocks in each module by addresses. Every N extended basic block in sequence form a group and R-

App.	Useful Gadgets(%)		Runtime O/H (%)	#RR	#Call/s	Memory O/H(MB)	App.	Useful Gadgets(%)		Runtime O/H (%)	#RR	#Call/s	Memory O/H(MB)
	Basic	Enhanced						Basic	Enhanced				
SPEC CPU2006 benchmark													
perlbench	10.69%	0.34%	12.42%	5	28235191	68.43	povray	10.16%	0.37%	11.51%	3	35031309	73.86
bzip2	9.58%	0.44%	2.12%	0	7297097	36.03	calculix	11.15%	0.29%	0.32%	4	3692203	88.97
gcc	10.37%	0.26%	11.50%	0	15710421	120.46	hmmer	10.55%	0.39%	-0.33%	0	1912030	47.44
bwaves	10.82%	0.33%	2.75%	0	259135	62.03	sjeng	9.69%	0.44%	9.21%	2	18088184	37.41
games	6.42%	0.15%	9.33%	424	3426329	167.50	GemsFDTD	10.56%	0.32%	-1.34%	0	1530312	68.50
mcf	9.59%	0.45%	2.20%	0	317371	35.26	libquantum	10.22%	0.40%	10.14%	1	74	42.25
milc	10.34%	0.40%	1.29%	1	394280	43.33	h264ref	8.87%	0.32%	3.86%	7	7201183	49.43
zeusmp	10.79%	0.32%	1.92%	0	1489	66.11	tonto	10.48%	0.23%	2.55%	8	8941394	110.98
gromacs	8.44%	0.30%	1.56%	0	930197	54.87	lbm	10.19%	0.41%	2.47%	0	3104	41.65
cactusADM	11.07%	0.32%	6.89%	0	4912	73.44	omnetpp	9.04%	0.37%	7.19%	0	11450543	70.44
leslie3d	12.45%	0.38%	1.78%	0	85206	63.01	astar	9.67%	0.37%	-0.94%	0	124959	54.85
namd	9.54%	0.40%	1.82%	0	741924	58.59	wrf	9.40%	0.22%	-0.86%	0	2158017	111.79
gobmk	12.50%	0.36%	14.36%	541	14029556	60.96	sphinx3	9.91%	0.40%	8.48%	120	1819791	45.05
dealII	8.65%	0.30%	14.08%	0	19291456	133.44	xalancbmk	19.40%	0.42%	16.23%	0	25047805	169.83
soplex	9.04%	0.37%	2.16%	2	302930	65.17	**Average**	**10.26%**	**0.35%**	**5.33%**	–	–	**43.41**
Parsec-2.1 benchmark													
blackscholes	10.56%	0.40%	14.36%	1	14663214	42.86	fluidanimate	9.97%	0.43%	1.86%	2	78180	56.73
bodytrack	9.13%	0.34%	23.78%	508	7028265	66.91	vips	13.50%	0.32%	2.63%	3	857260	125.35
facesim	10.87%	0.39%	10.11%	105	8498220	110.84	x264	10.72%	0.32%	24.19%	505	24081395	52.66
ferret	7.54%	0.23%	21.51%	3109	4637120	112.95	canneal	9.86%	0.42%	6.35%	28	2173629	57.08
freqmine	9.09%	0.45%	1.84%	1	653920	49.86	dedup	9.90%	0.43%	1.94%	1	1305985	38.30
raytrace	3.39%	0.07%	8.11%	1	3716974	156.61	streamcluster	9.86%	0.42%	2.97%	1	121403	56.83
swaptions	9.75%	0.42%	5.71%	1	21253310	57.32	**Average**	**9.55%**	**0.36%**	**9.64%**	–	–	**75.72**
Web servers													
Nginx	9.80%	0.29%	–	–	–	63.28	Apache	11.28%	0.33%	–	–	–	75.30

Table 3: Statistical data of ReRanz when applying to apps. #RR is the number of re-randomization when threshold = 0.

eRanz randomizes the order of the extended basic blocks in each group. The start addresses of these extended basic blocks are also random. To increase entropy, the shuffling process places a random number (less than M) of '*0xd6*'[7] between adjacent extended basic blocks. In section 6, we will study the effect of N and M on the performance.

6. Evaluation

We implement ReRanz on Ubuntu 12.04/Intel E74807 machine with 48 cores, 1.6GHZ, and 16GB RAM. In our experiment, we use SPEC CPU2006 [23], the multi-threaded Parsec-2.1 [9], Apache server httpd-2.4.1 [1] and Nginx 1.4.0 [5] as benchmarks. The Apache web server is configured to work in the *mpm-worker* mode with 16 threads. Nginx is configured to work with 4 worker processes.

6.1 Security Effectiveness

6.1.1 Memory Snapshot Analysis

We measure the gadgets obtained from the code variants to evaluate how effective can ReRanz fence off *code reuse* attacks. Our method is to dump a code variant for each application in Table 3 randomly at runtime and use *ROPgadget-5.4* [36] tool to find *useful gadget*s in these code variants. The scan depth (i.e., the max length of a gadget) of the tool is set to 100-bytes (default is 10). We evaluate the gadgets from the following two aspects. a. The ratio of *useful gadget*s (mentioned in section 4.2.1) in our basic design (Column "Basic"); b. The ratio of *useful gadget*s in the enhanced design (Column "Enhanced"). From Table 3, we can see that compared with the basic design, the enhanced design can eliminate (>99.6%) more *useful gadget*s. The *useful gadget*s are mainly from *libc.so* and *libstdc++.so* libraries' code variants. We also extracted the *useful gadget*s for each application and used the build-in gadget compiler of the tool to craft a *code reuse* attack that causes the program to start a shell. The results show that they all failed.

To further evaluate the categories of the *useful gadget*s, we extract and analyse all unique *useful gadget*s from all applications in Table 3. To be generic, we use the commonly classification proposed in [12, 13, 34]. The classification divides gadgets into six types, as shown in Table 4. When we increased the gadget length, the number of gadgets in each category also increased. However, the *System Call Gadget*s are consistently missing. We also analyzed all 14 Branch Gadgets, which conditionally change the stack pointer in the table, and found that they are all unusable. The reason is that even if these gadgets change the stack pointer, but the jump targets (next gadget) will not be affected by the stack pointer. So it would be very difficult to use the extracted gadgets to

Length \ Type	Load	Store	Arithmetic	Logic	Branch	Syscall
$\leq \infty$	685	275	146	38	14	0
≤ 15	263	106	84	15	0	0
≤ 10	108	36	25	0	0	0
≤ 5	80	0	19	0	0	0

Table 4: The category of gadgets in Table 3. Length denotes the number of instructions in a gadget.

[7] On the X86_64 platform, the encoding of '0xd6' represents an invalid instruction.

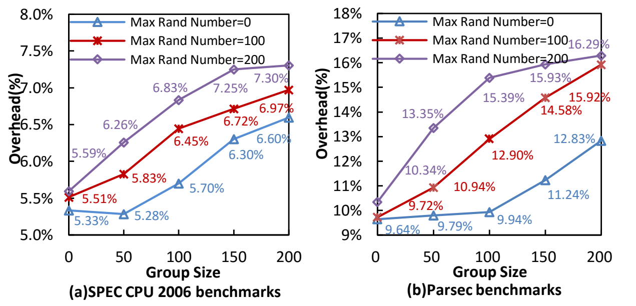


Figure 8: The impact of max random number and group size on SPEC CPU2006 and Parsec benchmarks.

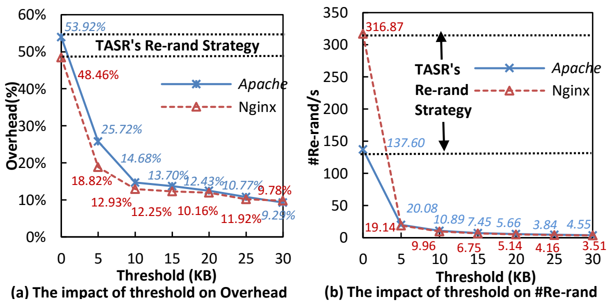


Figure 9: The performance of web servers.

launch a meaningful *code reuse* attack. Furthermore, Vasilis et al. observed that the actual gadgets used in real exploits have less than six instructions [33]. As shown in Table 4, when the gadget length is ≤ 5, four out of six gadget types are missing, making it very difficult for the adversaries to launch a real *code reuse* attack successfully.

6.1.2 Nginx Memory Disclosure Attack

We use Nginx 1.4.0 with a stack buffer overflow vulnerability to evaluate the effectiveness of ReRanz. This web server is popular and vulnerable to the Blind ROP attack. We use the optimized tool of Blind ROP [2] to conduct the attack against the Nginx server. This attack consists of three steps: Step1. **Stack Probing**: It overwrites the stack byte-by-byte to guess the real value of canary and the return address. If the worker process does not crash, the guessed value is correct; Step2. **Finding Enough Gadgets**: The tool scans the text segments by overwriting the obtained return addresses with pointers to each potential gadget entry and then inspecting the resulting program behaviors (e.g., *crash*, *block*, *closed* or *stays open*); Step3. **Build the Exploit**.

ReRanz defends against such attacks at all three steps. For the first step, it prevents obtaining the real return address. Even if the adversary obtained the real return address, it would still fail to find the *stop gadget* in the second step. A *stop gadget* is a necessary gadget to conduct this attack, whose behavior is to cause the program to block. Because the return address on the stack is not used by a *ret* instruction, when the attack overwrites the return address to probe a possible stop gadget, the worker process will never be blocked. As expected, the attack always fails.

6.2 Performance Evaluation

In the performance evaluation experiment, the shuffling process generates two code variants continuously. We evaluate ReRanz with different extended basic blocks' group sizes and max random numbers of padding bytes (i.e. '*0xd6*'). The re-randomization strategy is TASR's basic strategy (i.e., without output data accumulation). For the I/O intensive applications, Nginx and Apache servers, we evaluate the performance with different thresholds of accumulated data transmitted by *output* system calls.

6.2.1 SPEC CPU2006 and Parsec Benchmarks

Column "Runtime O/H" in Table 3 lists the runtime overhead of ReRanz on SPEC CPU2006 and Parsec benchmarks. The shuffling process only randomizes the start address of all extended basic blocks without reordering extended basic blocks and placing '*0xd6*'. The average overhead of SPEC is 5.33% and 9.64% for Parsec. Column "#RR" lists the number of re-randomizations and Column "#Calls/s" lists the invocation frequency of function calls in the original applications at runtime. From the table, we can observe that function calls are frequent in several benchmarks. As such, they manipulate RRAT frequently and cause high overhead. The frequency of re-randomization is high in *bodytrack*, *ferret* and *x264*, and thus high overhead.

Figure 8 shows the impact of the max random number of '*0xd6*' and the group size on SPEC and Parsec. Observe that the average runtime overhead increases with the growing size of groups and the growing number of '*0xd6*'.

6.2.2 Network I/O Performance Evaluation

We measure the performance of I/O intensive web servers using the same method as in [15, 30]. Particularly, we use the *ab* tool [7] on the client machine to send network requests (e.g., *ab −n 100000 −c 8 http://server-ip/index.html*) to retrieve the default work web page (612 bytes) of Nginx on the server running with ReRanz. The shuffling process only randomizes the start addresses of all extended basic blocks.

Figure 9(a) shows the performance of web servers with different thresholds of the accumulated amount of transmitted data. The performance overhead for TASR (threshold=0) is very high and both web servers suffer from decreasing overhead with the growth of the threshold. Figure 9(b) shows the frequency of re-randomization with different thresholds.

6.3 Memory Overhead

Because ReRanz uses shared memory, we measure the physical memory overhead. We achieve this by tracking the peak *Resident Set Size* (RSS[8]) of the protected application and its shuffling process over the entire program execution. Column "Memory O/H" in Table 3 presents the results

[8] It measures the size of process memory that remains resident in the RAM or physical memory. In the kernel module, using *get_mm_hiwater_rss(struct mm_struct *mm)* can get the value.

when the shuffling process only generates two code variants. For SPEC CPU2006 and Parsec benchmarks, ReRanz uses 73MB and 75MB additional memory on average. For web servers, the memory overhead is close to 69MB. Some applications in Table 3 have higher overhead because their code size and number of libraries used are larger. Such memory overhead is usually tolerable in modern systems.

7. Discussion

In this section, we review some security designs in ReRanz against the *memory disclosure* attacks. And then we discuss some limitations of ReRanz with our planned future work.

ReRanz needs to ensure the jump targets of all indirect branch instructions are the entries of basic blocks (i.e., the locations of trampolines), otherwise the protected process may run into undefined behaviour. Similarly, all recent fine-grained (basic block level) randomization or re-randomization [14, 17, 29, 44, 45] had the same presumptions. Differing from these techniques, ReRanz re-randomizes the basic blocks in the Code Cache instead of the original code region. This is to preserve the original code layout that is necessary for applications requiring *self-referencing* feature. Obviously, the adversaries could only utilize the *useful gadget*s, that named "basic block entry" gadgets in [29], to conduct the attack. Hence, ReRanz had enhanced the basic design from two aspects to reduce the number of *useful gadget*s. In both basic and enhanced designs, the adversaries could control the indirect branch instruction to jump to any location they want. For example, the jump target of an indirect call instruction = the value of the code pointer (stored in registers or memory) + *off_c* (shown in Fig. 3). For the adversaries, these code pointers are stored in the *payload* or the overflowed space (shown in Fig. 1). So the adversaries only need to tune the value of these pointers, and the indirect call instruction could jump to any locations (including the Code Cache). In the whole memory space, only the Code Cache has the executable permissionm but the code there is continuously re-randomized. So the adversaries cannot utilize the code of Code Cache directly. However, the adversaries could still utilize the remainder *useful gadgets* to conduct the *code reuse* attacks. But the number and the categories of these gadgets are very rare. So our enhanced design had raised the difficulty to use these gadgets to perform a real *code reuse* attack. But we should note that almost all of the remainder trampolines in the enhanced design are the function entries. So the adversaries could conduct the *function reuse* attacks [16], such as COOP [37] and Return-into-libc [6] attacks. In the future, we plan to further minimize the use of trampolines for functions in order to prevent against such attacks. ReRanz currently monitors the accumulated amount of transmitted data to trigger re-randomization. Watching over the content transmitted might help to make more accurate decisions on whether re-randomization is warranted.

Some techniques of ReRanz can also be applied to other virtual machines with dynamic binary translation to improve their security and performance.

8. Conclusion

ReRanz is a light-weight virtual machine using dynamic code re-randomization to mitigate *memoy disclosure* attacks. It leverages a helper process, called the "*shuffling process*", to generate code variants for the protected process through the *shared memory* mechanism. This approach preserves the DEP protection mechanism and effectively eliminates re-randomization caused delays. Unlike other binary re-randomization approach, ReRanz does not require expensive and difficult pointer tracking. Instead, it uses dynamic address translation to handle pointer related jumps. The translation procedure is hard to exploit by the adversary to launch a meaningful *code reuse* attacks. And our prototype can prevent from the Blind ROP attack against the popular web server, Nginx.

Acknowledgments

We would like to thank the anonymous reviewers for their useful feedback. This research is supported by the National Natural Science Foundation of China (NSFC) under grant 61332009 and 61303052, the Beijing Municipal Science & Technology Commission Program under grant D161100001216002. Author Wei-Chung Hsu is partially supported by MOST 105-2218-E-002-017 and MOST 105-2622-8-002-002.

References

[1] Apache HTTP Server. In *http://httpd.apache.org/*.

[2] Blind ROP tool. In *http://www.scs.stanford.edu/brop/*.

[3] LLVM Compiler Infrastructure. In *http://llvm.org/*.

[4] Libunwind library. In *http://www.nongnu.org/libunwind/*.

[5] Nginx Web Server. In *http://nginx.org/*.

[6] Getting around non-executable stack (and fix). In *http://seclists.org/bugtraq/1997/Aug/63*.

[7] ab tool. In *https://httpd.apache.org/docs/2.4/programs/ab.html*.

[8] M. Backes and S. Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 433–447, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7.

[9] C. Bienia and K. Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[10] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 268–279, New York, NY, USA, 2015. ACM.

[11] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazires, and D. Boneh. Hacking Blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242, May 2014.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[13] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[14] Y. Chen, Z. Wang, D. Whalley, and L. Lu. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 50–61, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3935-3.

[15] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *NDSS*. The Internet Society, 2014.

[16] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 243–255, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5.

[17] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for X86 and ARM. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[18] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[19] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 40–40, Berkeley, CA, USA, 2012. USENIX Association.

[20] E. G?ktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, May 2014. doi: 10.1109/SP.2014.43.

[21] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining Information Hiding (and What to Do about It). In *25th USENIX Security Symposium (USENIX Security 16)*, pages 105–119, Austin, TX, Aug. 2016. USENIX Association. ISBN 978-1-931971-32-4.

[22] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 368–379, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978356. URL `http://doi.acm.org/10.1145/2976749.2978356`.

[23] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006. ISSN 0163-5964.

[24] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd My Gadgets Go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585, May 2012.

[25] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, Washington, D.C., Aug. 2015. USENIX Association.

[26] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205, May 2013. doi: 10.1109/SP.2013.23.

[27] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978321. URL `http://doi.acm.org/10.1145/2976749.2978321`.

[28] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC*, pages 339–348. IEEE Computer Society, 2006.

[29] H. Koo and M. Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 23–34, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4233-9.

[30] K. Lu, S. Nürnberger, M. Backes, and W. Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *23rd Annual Symposium on Network and Distributed System Security (NDSS 2016)*, 2015.

[31] Microsoft. Data Execution Prevention (DEP).

[32] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615, May 2012.

[33] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, Washington, D.C., 2013. USENIX. ISBN 978-1-931971-03-4.

[34] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012. ISSN 1094-9224.

[35] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically Returning to Randomized lib(c). In *ACSAC*, pages 60–69. IEEE Computer Society, 2009. ISBN 978-0-7695-3919-5.

[36] J. Salwan. ROPGadget. In *http://shell-storm.org/project/ROPgadget*.

[37] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, May 2015.

[38] J. Seibert, H. Okhravi, and E. Söderström. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 54–65, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6.

[39] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM. ISBN 1-58113-961-6.

[40] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588, May 2013.

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

[42] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar. Eternal war in memory. *IEEE Security Privacy*, 12(3):45–53, May 2014. ISSN 1540-7993. doi: 10.1109/MSP.2014.44.

[43] U.Wiki. Address space layout randomization (ASLR).

[44] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4.

[45] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 367–382, GA, Nov. 2016. USENIX Association.