Quantitative Assessment on the Limitations of Code Randomization for Legacy Binaries

Pei Wang*†
Independent Researcher
Santa Clara, USA
uraj@apache.org

Jinquan Zhang
The Pennsylvania State University
University Park, USA
jxz372@psu.edu

Shuai Wang†
HKUST
Hong Kong, China
shuaiw@cse.ust.hk

Dinghao Wu
The Pennsylvania State University
University Park, USA
dinghao@psu.edu

***Abstract*—Software development and deployment are generally fast-pacing practices, yet to date there is still a significant amount of legacy software running in various critical industries with years or even decades of lifespans. As the source code of some legacy software became unavailable, it is difficult for maintainers to actively patch the vulnerabilities, leaving the outdated binaries appealing targets of advanced security attacks. One of the most powerful attacks today is code reuse, a technique that can circumvent most existing system-level security facilities. While there have been various countermeasures against code reuse, applying them to sourceless software appears to be exceptionally challenging.**

Fine-grained code randomization is considered to be an effective strategy to impede modern code-reuse attacks. To apply it to legacy software, a technique called binary rewriting is employed to directly reconstruct binaries without symbol or relocation information. However, we found that current rewriting-based randomization techniques, regardless of their designs and implementations, share a common security defect such that the randomized binaries may remain vulnerable in certain cases.

Indeed, our finding does not invalidate fine-grained code randomization as a meaningful defense against code reuse attacks, for it significantly raises the bar for exploits to be successful. Nevertheless, it is critical for the maintainers of legacy software systems to be aware of this problem and obtain a quantitative assessment of the risks in adopting a potentially incomprehensive defense. In this paper, we conducted a systematic investigation into the effectiveness of randomization techniques designed for hardening outdated binaries. We studied various state-of-the-art fine-grained randomization tools, confirming that all of them can leave a certain part of the retrofitted binary code still reusable. To quantify the risks, we proposed a set of concrete criteria to classify gadgets immune to rewriting-based randomization and investigated their availability and capability.

***Index Terms*—legacy software, code-reuse attack, code randomization, binary rewriting, risk assessment**

1. Introduction

To date, there are still many critical legacy software systems serving the aviation industry, the healthcare industry, governments, military agencies, and financial institutes [11], [8]. Many of these systems already have 15 to 30 years of life span [6], and the failures of these systems can easily cause considerably severe financial losses. An example of still active legacy software is DECOR, a Windows 3.1 program used to assist airplane takeoff and landing. In 2015, a failure of DECOR caused an airport in Paris to shut off for a day [12].

Maintaining legacy software systems and hardening them to resist emerging security threats is extremely important for both software vendors and users. As more and more sophisticated cyberattack methods are being invented, legacy software has become increasingly attractive targets of those attacks. One of the most threatening attacks is code reuse, a technique that allows attackers to execute malicious code with the victim programs' own constructs. Primitive defenses such as coarse-grained binary randomization (i.e., Address Space Layout Randomization [10]) provided by modern operating systems are no longer adequate to impede these attacks [72], [41]. Therefore, techniques that randomize binaries at a more fine-grained level are gaining more and more attention. DARPA's Cyber Fault-tolerant Attack Recovery (CFAR) project has shown a special interest in protecting legacy binaries by randomization [1] with millions of dollars of investment [5].

Despite that fine-grained randomization is in general much more capable of preventing advanced code reuse attacks, the defense is not directly applicable to legacy software systems, for their source code is either unavailable or incompatible with modern compilers. In certain cases, maintainers may not even be able to confirm or trust the correlation between the historic source code and deployed legacy binaries. As such, one of the few options left for maintainers is to directly reconstruct the legacy binaries without source code. Many fine-grained code randomization techniques have been proposed on the basis of binary rewriting such that legacy software can be hardened without source code or other auxiliary information [48], [80], [63], [38], [30]. However, we noticed that these *rewriting-based* randomization techniques can be problematic due to some fundamental technical challenges.

One of the major difficulties originates from the fact that for many legacy binaries, their program elements became *unrelocatable* at the time when the binaries were linked and stripped, meaning the addresses of these program elements cannot be easily changed. To circumvent

*Pei Wang is now a software engineer at Google LLC.
†Most of the work was done when the authors were graduate students at The Pennsylvania State Unviersity.

this problem, rewriting-based randomization techniques rely on address translation mechanisms which map the original addresses of randomized targets to their new addresses at run time. These mechanisms make it feasible to randomize binary code at a much finer granularity without relocation information, but certain parts of the retrofitted binaries conceptually remain unrandomized because the address translation is also available to code-reuse attackers. At this point, technical barriers make this translation indispensable to correctness. By exploiting those address translation mechanisms, attackers can access a subset of program elements in the randomized copies with their original addresses.

We believe that the risks imposed by the aforementioned problem should not be neglected. We have analyzed various existing fine-grained randomization defenses based on legacy binary rewriting and confirmed that they are truly vulnerable to code reuse attacks, regardless of the design and implementation of each technique. To help maintainers of legacy software better understand the potential hazard of deploying an impotent defense, we propose a systematic method to quantitatively assess the chance of a rewritten and randomized binary still being exploitable by code-reuse attacks. The assessment results allow security engineers to make more informed decisions when they seek to protect their legacy software systems.

In summary, we made the following contributions in this research,

- We are the first to identify and systematically analyze a critical security defect carried by fine-grained code randomization techniques built on top of legacy binary rewriting. With this defect, the randomized binaries may remain vulnerable to code-reuse attacks, even if the randomization granularity is refined to the instruction level. We are the first to investigate this long overlooked problem in depth.
- We developed a systematic methodology to quantitatively assess the potential risks of deploying fine-grained randomization defenses for legacy binaries against code-reuse attacks.
- We applied our risk assessment to the DARPA Cyber Grand Challenge binaries and several widely deployed real-world applications. Our assessment indicates that the effectiveness of rewriting-based randomization can significantly differ for different binaries, suggesting that it is crucial to understand the potential risks before such defenses are deployed.

For the remainder of the paper, we first introduce relevant background knowledge in Section 2. We then explain in detail why fine-grained randomization is vulnerable if based on legacy binary rewriting in Section 3. The threat model is in Section 4. Section 5 presents a classification for usable gadgets in regards to the defect of rewriting-based randomization. Section 6 reviews previously proposed code-reuse attacks and examine whether they conform our threat model and criteria for usable gadgets. The methodology for conducting the risk assessment is introduced in Section 7, followed by the assessment results for a set of widely deployed real-world binaries in Section 8. Section 9 discusses some important topics related to the assessment. We review related research in Section 10 and conclude the paper in Section 11.

2. Technical Background

2.1. Code-Reuse Attacks

Code reuse is an attack technique that aims to perform malicious computation within a program by exploiting memory vulnerabilities like buffer overflow and hijacking the normal control flows. Different from code-injection attacks, code-reuse attacks do not need to place new instructions into the program's address space, thus cannot be stopped by Data Execution Prevention (DEP).

A code-reuse attack is commenced by executing a chain of the victim program's own code snippets, each of which ends with an indirect control transfer instruction. The target addresses of these instructions are controlled by attackers so that the execution follows an unintended path crafted for malicious purposes. Each reused code snippet is called a *gadget*. For hardware architectures that encode instructions with varied lengths of bytes, e.g., x86, the address of instruction fetching is not forced to be aligned. Therefore, it is possible to interpret the memory byte sequence from an offset that is not intended to be the beginning of an legit instruction. A gadget obtained in this way is called an *unintended gadget*. Otherwise, it is intended. Early code-reuse attacks use unintended gadgets [71], but it was later revealed that they are not necessary [25], [58],

On x86, there are three kinds of indirect transfers, i.e., function returns, indirect branches, and indirect calls. Depending on which kind of instructions are used to chain different gadgets, the corresponding code-reuse scheme is called return-oriented programming (ROP) [71], jump-oriented programming (JOP) [28], [22], or called call-oriented programming (COP) [46], respectively.

2.2. Code Randomization

To hinder code-reuse attacks, code randomization is proposed to make the addresses of instructions unpredictable at run time, such that attackers cannot effectively construct the attack payload to connect each gadget.

Coarse-grained code randomization [10] permutes gadget addresses by loading code modules into non-deterministic locations in the address space. In contrast, fine-grained randomization [45], [44], [80], [48], [63] not only mutates gadget addresses with more entropy but also destroys certain gadgets. Therefore, fine-grained randomization is much more difficult to compromise. Basic methods of fine-grained randomization include function reordering, basic block reordering, instruction reordering, and register reassignment. There are two major approaches to achieving fine-grained randomization—by specialized compilation [19], [53], [49], [35], [20], [15] and by binary rewriting [48], [80], [63], [38]. Some fine-grained randomization takes place at load time, but still requires support from compilers [53], [15].

3. Limitations of Current Defenses

3.1. Fundamental Cause

The re-engineering of legacy binaries faces many challenges, such as binary disassembly and function slicing [17], [82]. One of the most fundamental challenges

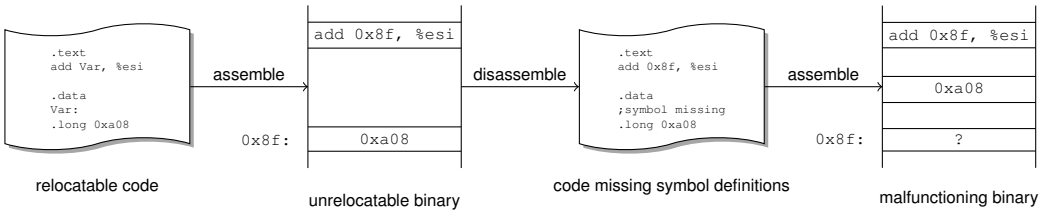

Figure 1. The relocation problem in binary rewriting.

is the lack of relocation information, as discussed by Wang et al. [79] and Wang et al. [78]. They indicated that one of the major obstacles forbidding straightforward binary reconstruction is the lack of relocation information in legacy binaries.

The problem can be illustrated by Figure 1, which shows a case of trivial binary rewriting. In the example, a binary is disassembled and immediately reassembled back without any semantics modification. What makes this process flawed is that during the disassembly process, the first operand of the `add` instruction, i.e., the concrete address `0x8f`, is not lifted to a symbolic value. Since there is no guarantee that the linker will keep the data at address `0x8f` when reassembling the binary, the same `add` instruction in the reassembled copy will likely fetch the wrong data.

While it is not difficult to fix the error in Figure 1 caused by unrelocatable disassembly, the problem is exceedingly challenging in general. Recovering the relocation information is equivalent to solving the following type inference problem: given a pointer-size data chunk in the binary, the rewriting algorithm needs to decide whether it is used as a pointer. For binary rewriting, the type inference problem has to be solved with both soundness and completeness to ensure the rewritten binary does not malfunction. In this context, soundness means no pointers are inferred as non-pointers and completeness means all inferred pointers are actually pointers.

Historically, researchers [85], [84] have developed various heuristics to recognize code pointers in the data sections of stripped binaries, some of them achieving 100% accuracy for certain binaries [79], [78]. However, all of existing methods in theory can pledge only soundness but not completeness, thus the correctness cannot be assured.

3.2. Address Translation Mechanisms

By the time of paper writing, all rewriting-based randomization techniques known to us circumvent the relocation problem by introducing run-time *address translation* mechanisms. The basic idea is to leave binary data unmodified and instrument indirect control-flow transfer instructions so that the transfer target can be redirected to the correct address. In this way, rewriters are only required to identify code pointers with soundness but not completeness. Address translation in rewritten binaries can be abstracted as Figure 2. We argue that *this address translation mechanism is exactly the Achilles' heel of rewriting-based randomization.* Regardless of the design and implementation of a rewriting-based randomization technique, the address translation mechanism has to be bundled into the rewritten binary. By exploiting the translation as a

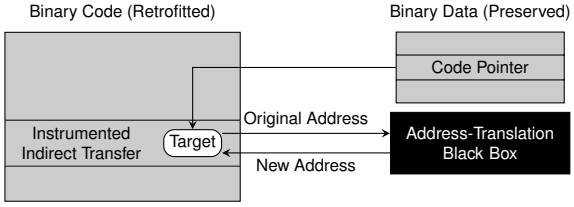

Figure 2. Abstraction of address translation mechanisms in binary rewriting.

black box, attackers can reliably locate certain gadgets in randomized binaries by inspecting their addresses in unrandomized copies.

3.3. Examples

To deliver a concrete understanding of the address translation mechanisms discussed above, we introduce two examples of fine-grained randomization techniques based on binary rewriting. We use two examples to illustrate the security issue discussed above.

3.3.1. Static Randomization: Binary Stirring. Binary stirring [80] is a representative fine-grained randomization technique that supports the hardening of legacy binaries. Binary stirring addresses the relocation issue by reserving the address space occupied by the original binary code sections and placing "stubs" at a set of addresses which overapproximates the set of all indirect control-flow transfer targets in the program. Take Figure 3 as an example. A segment of the original binary is in Figure 3a and the corresponding segment of the stirred binary is in Figure 3b. The stirred binary has two parts—the `.told` section which lives in the exactly same address space as `.text`, and the `.tnew` section which is a randomized version of `.text`.

Of these two parts, `.tnew` is the code that will be actually executed, while `.told` now serves as read-only data. Binary stirring makes three notable changes to the binary in order to preserve the original semantic:

1) Modify the targets of direct control flow transfers, i.e., the operand of the `jmp` instruction (framed in Figure 3b). [1]
2) Replace the bytes at the original address of a potential transfer target with a special stub byte `f4` (the opcode of `halt` that should not appear in normal programs) followed by the little-endian encoding of the randomized address of that target. In the example of Figure 3,

1. In this example the encoding of the instruction is unmodified because the near `jmp` uses a PC-relative address as its operand.

```
.text:
080483f3: c7 45 f8 0a 84 04 08  mov   $0x804840a,-0x8(%ebp)
080483fa: c7 45 fc 13 84 04 08  mov   $0x8048413,-0x4(%ebp)
08048401: 8b 45 08              mov   0x8(%ebp),%eax
08048404: 8b 44 85 f8           mov   -0x8(%ebp,%eax,4),%eax
08048408: ff e0                 jmp   *%eax
0804840a: 83 45 0c 04           add   $0x4,0xc(%ebp)
0804840e: 8b 45 0c              mov   0xc(%ebp),%eax
08048411: eb 07                 jmp   0x804841a
08048413: 8b 45 0c 05           add   $0x5,0xc(%ebp)
08048417: 8b 45 0c              mov   0xc(%ebp),%eax
0804841a: c3                    ret
```

(a) Original binary

```
.told:
080483f3: c7 45 f8 0a 84 04 08 c7 45 fc 13 84 04 08 8b 45
08048403: 08 8b 44 85 f8 ff e0 f4 11 84 04 09 45 0c eb 07
08048413: f4 1a 84 04 09 45 0c c3

.tnew:
090483f3: c7 45 f8 0a 84 04 09  mov   $0x804840a,-0x8(%ebp)
090483fa: c7 45 fc 13 84 04 09  mov   $0x8048413,-0x4(%ebp)
09048401: 8b 45 08              mov   0x8(%ebp),%eax
09048404: 8b 44 85 f8           mov   -0x8(%ebp,%eax,4),%eax
09048408: 80 38 f4              cmpb  $0xf4,(%eax)
0904840b: 0f 44 40 01           cmove 0x1(%eax),%eax
0904840f: ff e0                 jmp   *%eax
09048411: 8b 45 0c 05           add   $0x5,0xc(%ebp)
09048415: 8b 45 0c              mov   0xc(%ebp),%eax
09048418: eb 07                 jmp   0x9048421
0904841a: 83 45 0c 04           add   $0x4,0xc(%ebp)
0904841e: 8b 45 0c              mov   0xc(%ebp),%eax
09048421: c3                    ret
```

(b) Stirred binary

Figure 3. An example of the address translation mechanism in binary stirring.

the two wavily underlined immediate numbers are recognized as potential indirect transfer targets. Two stubs are inserted at the addresses indicated by the underlined bytes in `.told`.

3) Instrument every indirect jump and indirect call instruction by inserting two instructions ahead, as underlined in Figure 3b. These two instructions check the first byte at the control-flow transfer target; if it is `f4` then replace the target with the value stored right after the `f4` stub. In this way, the control flow will be correctly directed to the randomized addresses.

Note that the set of inserted `f4` stubs is precisely the pre-computed mapping from original addresses to randomized addresses. This technique is called "trampolining" and has been widely adopted by previous research on binary retrofitting [48], [85], [39].

3.3.2. Static and Dynamic Randomization: Isomeron. Isomeron [38] is another example of rewriting-based fine-grained randomization. Different from binary stirring that provides static randomness, Isomeron introduces randomness dynamically throughout the program's lifetime. Binaries protected by Isomeron have two copies of code sections, one of which is the original and the other is processed by in-place randomization [63]. Isomeron employs the dynamic instrumentation system Pin [59] to trap every indirect control-flow transfer. Whenever such a transfer occurs, Isomeron randomly redirects the target to one of the two code copies. This makes the traditional gadget chaining techniques almost surely fail, because the chance that a code-reuse attack follows the designed sequence decreases exponentially with respect to the number of gadgets used.

TABLE 1. TOOLS AND TECHNIQUES EMPLOYING RUN-TIME ADDRESS TRANSLATION

Tool/Technique	Translation Type
In-Place Randomization [63]	Identity
Remix [31]	Identity
Isomeron [38]	Linear
CodeArmor [30]	Linear
ILR [48]	Non-Linear
Binary Stirring [80]	Non-Linear
Reins [81]	Non-Linear
MULTIVERSE [18]	Non-Linear

To implement such run-time behavior, Isomeron needs an address translation mechanism that maps the target addresses in one copy of the code to the corresponding addresses in the other copy. Isomeron employs in-place randomization and the offset between the corresponding basic blocks in the two copies is a constant. As such, the translation can be made very efficient. Note that this address translation is not specific to Pin. Switching to other similar systems like Dynamo [16] does not resolve the issue.

3.4. Validation of Analysis

Despite holding different views on the severity, our analysis on the defects of rewriting-based randomization is in alignment with the conclusions of some previous work [48], [80], [30]. For further validation, we manually applied the idea of binary stirring[2] to a small binary compiled from a vulnerable C program written by ourselves. Regarding this binary, we have composed a code-reuse exploit as a minimal working example to demonstrate the impact of the vulnerability.

3.5. Affected Defenses

Table 1 is a list of security tools and techniques featuring run-time address translations, thus affected by the problem discussed above. As far as we know, there are three types of address translation mechanisms. The identity translation is more of a theoretical concept for lacking an actual implementation, which is employed by randomization techniques that do not mutate basic block addresses [63]. The linear translation is for techniques that randomize the address of the entire code section as a whole [38], [30]. The most flexible translation type is the non-linear one, which can implement arbitrary address mappings [48], [80].

4. Threat Model and Assumptions

Our research aims to undertake an in-depth assessment of the potential risks caused by the previously discussed vulnerability. We first describe a threat model to regulate factors that need to be considered by the assessment.

2. Neither binary stirring nor Isomeron is available for evaluation. While we are not able to perform real-world tests on these systems, our analysis shows that any method employing the address translation mechanism is subject to this vulnerability.

Legacy Binaries. All defenses considered in the assessment must be designed for legacy binaries with only minimal symbol and relocation information. Indeed, releasing fully relocatable binaries becomes more and more common in recent years for the benefits of ASLR on modern operating systems [83]. However, as we emphasized in the introduction, legacy systems with unrelocatable binaries are still running in production environments, despite that the source code of these systems are no longer available or maintainable.

Code Randomization. We suppose binaries are protected by a fine-grained code randomization technique providing strictly better defenses than ASLR. The randomization is able to shuffle the layout of binary code at the function [45], [19], [53], basic block [80], and instruction level [48], [63]. It can also alter the content of the binary code with semantics-equivalent transformations [63]. This randomization technique should be based on legacy binary rewriting and must not require program source code or binary relocation information. Some research on binary randomization assumes that binaries contain relocation information [73], [15], [55], [65], [82], leaving them out of our consideration.

Adversary Capabilities. We assume that typical protections provided by a modern operating system are available and activated. Particularly, data execution prevention (DEP) is in effect and attackers cannot directly launch code-injection attacks. For exploit initiation, attackers should be able to find certain types of memory vulnerabilities to start with. There are many defensive techniques preventing attackers from exercising memory corruption so that code-reuse attacks are nipped in the bud, but it is widely believed that no mitigation, to date, is fully comprehensive with acceptable cost for practical deployment [52], [76]. We assume that the attacked programs contain vulnerabilities that allow attackers to inject malicious payload at the desired address in the memory and manipulate certain memory content so that the target of an indirect function call can be hijacked. These vulnerabilities are generally necessitated by code-reuse attacks based on call-oriented programming [46], [68], [43]. Additionally, we assume attackers can statically inspect or guess the layout of an unrandomized copy of the binary they aim to attack. However, attackers are *not* allowed to disclose binary code layouts at run time. Such attempts can be thwarted by defensive techniques that prevent code disclosure [14], [38], [30].

Control-Flow Integrity. Control-flow integrity (CFI) prevents indirect transfers that are not intended by normal program execution. A shadow stack, which is a particular kind of CFI enforcement, records every function call and return to make sure a function always returns to its caller [13]. CFI is essentially a category of defenses different from code randomization. Since our analysis focuses on fine-grained randomization alone, we assume that there is *no* dedicated CFI protection deployed; however, we later (in Section 5.2 and Section 5.3) show that the defenses considered by our threat model possess a probabilistic defensive effect equivalent to coarse-grained CFI *and* a shadow stack. In recent literature and this paper, being coarse-grained means the CFI policy maintains a universal set of valid targets for all call sites.

5. Gadget Analysis

Gadgets are the basic building blocks of code reuse attacks. Fine-grained code randomization blocks code-reuse attacks by eliminating code sequences that can be potentially used as gadgets, or making the addresses of such sequences unpredictable. We hereby propose three criteria to measure the usability of gadgets.

- *Stability* indicates whether gadgets can be preserved after randomizing transformations are applied.
- *Trackability* indicates whether attackers can still locate the gadgets after randomization.
- *Connectivity* describes a gadget's capability of transferring the control flow to the next gadget.

By definition, connectivity depends on the trackability of other gadgets. We call gadgets with stability and trackability the *Randomization-Resilient* (RR) gadgets. If an RR gadget also has connectivity, it is called an RR* gadget.

5.1. Gadget Stability

In our consideration, the key to gadget stability is whether the high-level semantics of the gadget are retained. To find out how gadgets can withstand randomization transformations, we reviewed existing fine-grained randomization implementations. In general, the transformations can be classified into the following categories based on the smallest program elements to which the transformations can be applied:

- *Instruction transformation* which randomizes individual instructions, e.g., atomic instruction substitution [67], [63].
- *Basic block transformation* that preserves basic block semantics, e.g., basic block address shuffling [45], [80] and instruction reordering within basic blocks [23], [63].
- *Function transformation* that may alter the semantics of individual basic blocks but preserve the semantics of functions, e.g., register preservation code reordering [63] and register reassignment [33], [66], [63], [35], [24]

The three types of transformations can be stacked together. Theoretically, it is also possible to perform inter-procedural transformations that randomize the application binary interfaces (ABI). To the best of our knowledge, such transformations, if there exists any, are not widely available at this point. Therefore, we do not consider randomization transformations that manipulate program semantics across function boundaries.

Instruction transformations are effective against unintended gadgets which strictly depends on instruction encoding. Basic block transformations have a high chance to eliminate intended gadgets that are incomplete fragments of basic blocks. Function-level transformations further destroy gadgets that do not confine the boundaries of inter-procedural control-flow transfers, i.e., function entries (invocations) and function exits (returns).

It should be noted that, besides the constraints above, *whether a gadget is stable also depends on how it is used in the attack chain.* If the attacker tries to reuse the high-level semantics intended by the source code from

which the gadget is compiled, stability will be guaranteed. However, in case the attacker reuses a gadget for its machine-level semantics or certain side effects, e.g., moving a particular constant into a particular register, the stability cannot be preserved. For example, one of the function transformations, register preservation code reordering, randomizes the order of register save (`push`) and restore (`pop`) sequences in a function [63]. With this transformation applied, attacks relying on a particular sequence of push and pop operations instead of the high-level function semantics will fail. The same analysis applies to other existing transformations such as register reassignment. *In the rest of this paper, we assume it is the high-level semantics that attackers intend to reuse.*

5.2. Gadget Trackability

As explained in Section 3.1, rewriting-based randomization techniques develop address-translation techniques to dynamically redirect code pointers loaded from binary data because rewriters cannot correctly identify code pointers in binary images without source code. Considering the translation mechanism as a mathematical function, its domain is exactly the set of the pre-randomization addresses of all trackable gadgets.

In general, address translators can be activated only through indirect control-flow transfers because direct control transfer targets can be correctly relocated by current techniques. Additionally, state-of-the-art binary rewriters have been able to safely rewrite pointers in code sections of a binary in most practical cases. Therefore, it is sufficient for address-translation components to exclusively redirect the addresses of *legitimate indirect transfer targets whose addresses have possibly appeared in the binary data sections*. If the given address is not one of such kind, the behavior of the translator may be undefined and therefore unpredictable to attackers. In that sense, a control-flow integrity policy is enforced. This integrity, however, is coarse grained, because address translators treat all indirect call sites in the same manner.

Consider the examples in Section 3.3. Binary stirring decides whether an address should be translated by looking for the `f4` stubs and only legitimate indirect transfer targets identified by the rewriter are marked with the stubs. Without the `f4` stubs, the translation will not happen. For Isomeron, the translation can always be triggered, but it preserves the semantics only if the given address points to a legitimate indirect transfer target in the original binary.

5.3. Gadget Connectivity

Either return instructions [71] or indirect jump/call instructions [22], [46] can be used to connect different gadgets in code-reuse attacks. However, return-oriented programming (ROP) is not effective when attacking rewriting-based randomization, for the address-translation mechanism is not necessarily available for return instructions. The reason is that a return address is dynamically generated by the hardware before the control flow switches to another context. There is no need to translate return addresses since they already correctly point to the randomized return sites. From the perspective of attackers, this is equivalent to a shadow stack. Therefore, only gadgets ended with indirect jumps and calls have connectivity.

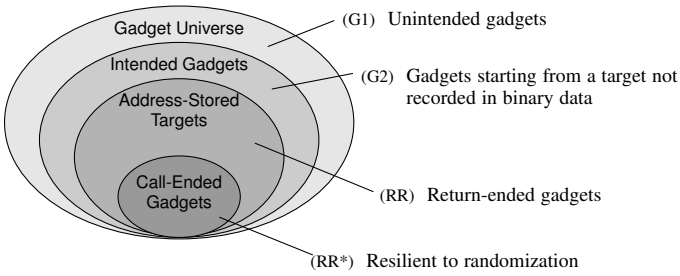

Figure 4. Gadget classification in code-reuse attacks. G1, G2, and RR refer to the differences between the enclosing and enclosed sets, respectively.

TABLE 2. GADGET RESILIENCE TO FINE-GRAINED RANDOMIZATION BASED ON BINARY REWRITING

Gadget Class	Stability	Trackability	Connectivity
G1			
G2	✓†		
RR	✓	✓	
RR*	✓	✓	✓

† G2 gadgets only have limited stability.

5.4. Summary

Based on the previous analysis, the universe of code-reuse gadgets can be divided into four classes, as shown in Figure 4. The characteristics of each class are summarized by Table 2.

G1 gadgets, i.e., the unintended ones, can be easily smashed by various randomization transformations. G2 gadgets are non-trackable gadgets starting from an instruction which is intended but not recognized as indirect targets by binary rewriters, including all instructions whose addresses are not stored in binary data sections. These gadgets can survive instruction transformations but not necessarily basic block and function transformations.

Randomization-resilient (RR) gadgets are those starting from legitimate function entries whose addresses can be found in binary data sections (address-stored targets) and ending with return instructions. RR gadgets are fully stable and trackable, meaning they can be located by attackers and reused for their retained semantics. However, RR gadgets lack connectivity since ROP is ineffective as explained in Section 5.3. For this reason, a RR gadget can only achieve "one step" during an attack. In case the attack goals are complicated and single-gadget attack is not feasible, attackers will need other methods to chain different RR gadgets together.

RR* gadgets are those same as RR gadgets in all aspects except that they end with indirect call instructions. RR* gadgets are completely stable, trackable, and can be used to connect other stable and trackable gadgets.

6. Effective Attacks

To better understand the risks caused by gadgets that are immune to rewriting-based randomization, we discuss

what code-reuse attack schemes can *fully* utilize them. If there exists such an attack, we can assess the risks by considering the feasibility of the attack regarding a particular binary. We emphasize that an attack scheme being feasible within our threat model does not mean it is suitable for risk assessment, because the conditions for launching such an attack may be too strict such that the actual risks may be under-estimated.

In the past, many code-reuse attack techniques have been proposed in response to different defensive measures. A natural research question is that whether any of these existing attacks fully matches the criteria of exploiting the vulnerability of rewriting-based randomization. We examined nine code-reuse techniques presented during the last two decades. *Our analysis indicates that none of them can be considered as the precise model of adversary when assessing the potential risks of rewriting-based randomization.* They either lead to over-estimation of the risks for utilizing gadgets that do not fit our usability criteria, or lead to under-estimation for excluding some of the usable gadgets in order to breach defenses other than code randomization. The analysis results are summarized by Table 3. This further suggests that the fundamental risks residing in current rewriting-based randomization techniques have not been systematically considered before.

The traditional ROP uses unintended gadgets and does not particularly take address-stored targets as gadget entries, thus breaking stability and trackability premises. It also relies on return instructions for gadget chaining, which violates connectivity as well. JOP is similar to ROP, except that it uses indirect jump for connecting gadgets.

The "Control-Flow Bending" [26] and StackDefiler attacks [34] partially rely on ROP, leading to the loss of gadget connectivity. Gadgets used by these two attacks likely lack trackability, but we do not have enough information to confirm it. The "Out of Control" attack [46] employs gadgets belonging to a super set of usable gadgets in our scenario and the proof-of-concept exploit presented in the paper lacks gadget connectivity. Like StackDefiler, the trackability of their gadgets is also questionable. The just-in-time (JIT) code reuse [74] and blind return-oriented programming (BROP) [21] need to read the code sections of the attacked binaries in order to search for usable gadgets at run time, thus hindered by disclosure prevention.

"Control Jujutsu" [43] is an attack exploiting the incompleteness of pointer alias analysis and the difference between the call graph enforced by a control-flow integrity policy and the actual call graph. "Control Jujutsu" is based on source code analysis and also uses call-oriented programming, so the gadgets employed respect stability and connectivity. Trackability, however, is not fulfilled because "Control Jujutsu" selects gadgets according to their reachability in call graphs which is a source code level property, while trackability requires all gadgets to be address-stored targets and can only be observed at the binary level. As shown by the NGINX case study in Section 8.3, at least one of the proof-of-concept attacks will be rejected by our threat model.

It has been known that programs written in object-oriented languages can be vulnerable to code-reuse attacks. By abusing dynamic function dispatching, attackers can conduct code reuse through injected malformed objects, using a technique called property-oriented programming (POP). POP has been developed for many programming languages, including Python [4], Java [56], and PHP [42], [37]. Counterfeit object-oriented programming (COOP) is a case of POP specialized for the C++ programming language. Although not directly applicable to C programs, COOP may fit all premises at its best effort when attacking C++ binaries. However, at least one of the exploits originally presented by COOP does not fulfill gadget trackability for directly employing a system function which is relocatable and thus cannot trigger address translation planted by binary rewriting. On the other hand, we have a case study on GCC (Section 8.3) showing that the infeasibility of COOP does not imply that the binary is safe from code-reuse exploits.

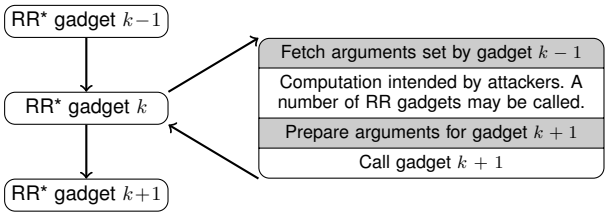

Figure 5. Basic RR Gadget chaining. Gray parts indicate operations for connectivity.

Given that none of the previously known attacks are suitable as benchmark for assessing the risk we discovered, we now describe an attack scheme that is a generalization of effective attacks on rewriting-based randomization. Similar to other memory-based exploits, we assume that the attack can be initiated by hijacking the control flow through an indirect function call with its target manipulated by attackers. Basically, an attack is a connected sequence of RR and RR* gadgets, among which the RR* gadgets are the key connecting hops. An RR* gadget can call some RR gadgets of their choice, but eventually the control flow is passed to the next RR* gadget. Conceptually, the code-reuse process should proceed in a manner illustrated by Figure 5. At the behavioral level, this way of chaining gadgets is similar to the so-called continuation-passing style (CPS) [75] in functional programming, which is theoretically as expressive as the direct or common programming style. Therefore, the lack of return instructions do not prevent attackers from performing complicated malicious tasks. Note that this is not the only feasible attack scheme. In certain cases, the connectivity of RR* gadgets are not necessary for launching a successful attack. We will further discuss this in Section 8.3.

7. Risk Assessment

Although rewriting-based randomization is unsound to prevent code-reuse attacks in general, it is possible that the defense is comprehensive for particular binaries. Therefore, a systematic approach to evaluating the potential risks of the vulnerability will be useful for deciding whether the defense should be adopted to protect a certain binary.

Constructing successful code-reuse attacks is heavily dependent on 1) exploiting one or multiple memory errors to initiate the exploit and 2) finding appropriate gadgets

TABLE 3. EFFECTIVENESS OF DIFFERENT CODE-REUSE ATTACKS AGAINST REWRITING-BASED FINE-GRAINED RANDOMIZATION

Attack	Effective Against	Disclosure Prevention	Gadget Stability	Gadget Trackability	Gadget Connectivity
Traditional ROP [71]	Data execution prevention		✗	✗	✗
JOP [22]	Shadow stack, return-less binaries		✗	✗	
Control-Flow Bending [26]	Fine-grained CFI without shadow stack			✗	✗
StackDefiler [34]	Fine-grained CFI and shadow stack			✗?	✗
Out of Control [46]	Coarse-grained CFI without shadow stack			✗?	✗
JIT Code Reuse [74]	Fine-grained randomization	✗			
BROP [21]	Conceptually fine-grained randomization	✗			
Control Jujutsu [43]	Fine-grained CFI and shadow stack			✗	
COOP [68]	Coarse-grained CFI and shadow stack			✗?	

✗ means the attack violates the premise by design or fails to consider it. ✗? indicates either i) the attack likely violates the premise but we are unable to verify it due to the lack of information, or ii) the attack may choose to respect the premise but one or more proof-of-concept exploits reported by the authors violate it.

to achieve the malicious purpose. As a clarification, evaluating the severity and impact of memory errors is out of the scope of this paper. Instead, our assessment focuses on measuring the *availability and capability* of gadgets immune to binary rewriting, i.e., RR and RR* gadgets introduced in Section 5.

It is true that the existence of usable gadgets alone does not always ensure successful exploits, since the success of code-reuse attacks also depends on many other factors. Nevertheless, gadget availability and capability are both indicators of confidence and prerequisites for further assessment. Since one single effective attack can be devastating enough, we always try to be conservative in the assessment and consider it dangerous to underestimate any potential risks.

7.1. Assess Gadget Availability

Our assessment takes a two-step approach when counting usable gadgets. The first step is to get the total number of RR gadgets and RR* gadgets, which are the only two kinds of gadgets having stability and trackability. Since the union of RR and RR* gadgets is essentially all address-stored functions, identifying them is equivalent to identifying code pointers in the data sections of the examined binary. Although there are different methods to excavate code pointers in legacy binaries, most of them are based on heuristics and can be inaccurate. In general, different randomization techniques make different over-approximations on the set of indirectly called targets.

Our assessment method is designed for scenarios where the deployed randomization technique can be either known or unknown. Therefore, we propose two strategies for counting RR and RR* gadgets. In case a known randomization technique is considered, the assessment can simply adopt the identification algorithm shipped with that technique. Otherwise, the assessment falls back to a default method widely used by previous work [85], [84], [79], [30]. Briefly, this method scans all the data sections of a binary and identifies all aligned pointer-size data as code pointers, as long as their values are within valid ranges of code sections and point to function entries.

The second step is to differentiate RR* gadgets from RR gadgets. We deem a call site in a gadget *steerable* if the target of that call site can be fully controlled by the input to the gadget. An RR* gadget is therefore an usable gadget with at least one steerable call site. We perform static analysis to probe steerable call sites in assessed binaries. Due to the inherent limitation of static analysis, we develop both an aggressive analysis strategy that potentially over-approximates the number of RR* gadgets and a conservative one that possibly leads to under-approximation.

7.1.1. Aggressive Assessment. We can aggressively count every address-stored function containing an indirect `call` instruction as an RR* gadget, yielding an upper bound for the assessment. In some scenarios, this method may be potent enough for estimating the availability of RR* gadgets. In others, however, it may significantly overestimate the potential risks. The reason is that, even if the target of a call site is indirectly invoked, attackers may not have full control over the destination. The code snippet below illustrates one of such cases.

```
1  const void (*fptr_array[2])() = {
2    func1, func2
3  };
4
5  void gadget(unsigned int arg) {
6    fptr_array[arg % 2]();
7  }
```

In this example, regardless of the value of the argument provided to the gadget function, the indirect call at line 6 can only invoke one of the two functions stored in the constant array defined at line 1.

7.1.2. Conservative Assessment. In case the assessment is expected to deliver a conservative estimate on the risks, we devise a program analysis algorithm to rule out such less capable gadgets.

Typically, a local value inside a gadget can be arbitrarily set if the data flow to this value transitively depends on attacker input only.[3] If attackers can provide arbitrary input to a gadget, the only values that cannot be fully controlled are hard-coded constants and values dependent on those constants. Therefore, we employ the reaching definition analysis to solve data dependencies. By constructing the def-use chain for all instructions in a function, we can solve the flow dependence problem which describes read-after-write (RAW) dependencies [62]. If with this analysis

3. We do not consider program invariants. An artificial example is "`b=a-a;`" where the value of `b` cannot be controlled through manipulating the value of `a`. Finding program invariants is at least as hard as detecting opaque predicates, which is still an open problem being actively researched [60].

we find an indirect call instruction transitively depends on a constant value, we conclude that attackers may not have full control over the target of this instruction, meaning this call site is not steerable in the conservative assessment setting.

For a sound reaching definition analysis, we need to handle memory aliasing. Since a definition may be stored into memory and later extracted by memory loads, it is imperative to find out which definitions a memory access may refer to. In our implementation, we assume any two memory accesses may refer to the same memory cell, which is extremely conservative but correct. We also need to derive the use and def sets for each x86 instruction, taking dependencies of CPU flags and implicitly used operands into consideration. Overall, our reaching definition analysis is sound, and typically leads to false positives. However, we use the analysis result in the way that whenever the definition of a constant value reaches a call site target, we disqualify this call site from being steerable. Therefore, all RR* gadgets identified by the analysis will indeed have full connectivity.

7.2. Assess Gadget Capability

In typical code-reuse attacks, a gadget is used either to connect another gadget or to maliciously modify the state of the victim program. As such, we consider the capability of a gadget in two aspects, corresponding to the two different usages.

For an RR* gadget, its capability of chaining other gadgets can be described by its *capacity*, defined as the number of steerable call sites inside that gadget. [4] Since each steerable call site can invoke another trackable gadget, the higher capacity an RR* gadget has, the more complicated behavior it may be used to implement. RR* gadgets with a non-trivial capacity, i.e., capacities greater than one, can greatly enrich the behavior of an attack. For example, if an RR* gadget includes a branch and both paths of the branch has a steerable call site, attackers may use this RR* gadget to implement the if-else logic. A single-path RR* gadget with a capacity of n can call at most $(n-1)$ RR gadgets before getting to the next RR* gadget. Note that the capacity of an RR* gadget can be decided either aggressively or conservatively, as defined in the previous subsection. Since whether a call site is steerable can be decided either aggressively or conservatively, the capacity of an RR* gadget is also subject to this difference.

Another dimension of the gadget capability is the quantity and variety of sensitive functions invoked by the gadgets. Typically, a code-reuse attack needs to invoke certain system-provided functions to manipulate program states. According to our observation, it is extremely rare that these functions are called through indirect control-flow transfers, meaning that the sensitive functions themselves are unlikely to be trackable gadgets. Therefore, attackers will need to use a RR or RR* gadget to invoke such sensitive functions. Deciding what functions are sensitive is subject to the analyzed binary, its deployment environment, and user requirement.

4. Under this definition, RR gadgets can be viewed as RR* gadgets with 0 capacity.

8. Experimental Assessment

To evaluate our risk assessment method, we implemented it as a binary analysis framework with about 6700 lines of Java and Scala code plus about 1200 lines of Python scripts. The implementation includes the default code pointer identification algorithm and the conservative data-flow analysis algorithm introduced in Section 7.1. The framework currently only supports the x86 architecture, which is the most common platform hosting legacy production binaries.[5] Based on this prototypical implementation, we assessed a set of carefully selected binaries.

8.1. Analyzed Binaries

Ideally, we should employ real-world legacy systems that are still in production as evaluation objects. However, such systems are mostly owned and operated by governments, military, and financial institutes and are typically not accessible to outsiders. Fortunately, DARPA recently launched an event called Cyber Grand Challenge (CGC) which is the first competition aiming to create automated systems that can analyze and patch the vulnerabilities of legacy software [2]. Throughout the CGC event, DARPA released a collection of challenge binaries which are custom-made programs designed to intentionally contain a wide spectrum of software flaws that may lead to security breaches, including buffer overflow, use after free, and type confusion, etc. The challenge binaries are meant to approximate real legacy software so that they can be used to evaluate existing binary analysis and transformation techniques. Our evaluation thus took the challenge binaries for experimental risk assessment.

In addition to binaries from the CGC competition, we also selected nine popular open source applications. Although these open source programs are neither legacy nor created to mimic legacy systems, they are more realistic than the CGC binaries in terms of code base scale and complexity, thus a meaningful complementary to the CGC binaries for evaluating our work.

8.1.1. Cyber Grand Challenge Binaries. Originally, the challenge binaries were developed for the DECREE system which is a simplified Linux variant. DECREE does not support threads, shared memories, or signals. Only seven system calls are available on DECREE, including `receive` and `transmit` for socket communication, `allocate` and `deallocate` for memory mapping management, `fdwait` for synchronous I/O multiplexing, `random` for accessing the system's entropy pool, and `_terminate` for halting program execution. A group of researchers have ported 241 challenge binaries to Linux and Mac OS X [3]. Our evaluation uses the ported binaries. A first look at these binaries showed that 34 of them contain indirect function calls, which is the premise of usable code-reuse gadgets being existent. As such, further analysis only considered these 34 binaries. The names and sizes of these binaries are listed in the first two columns of Table 4. The sizes of the analyzed challenge binaries vary from 16 KB to 19272 KB, with the average being

5. With additional engineering effort, the framework can be extended to support other hardware architectures.

TABLE 4. ANALYZED CGC BINARIES AND GADGET STATISTICS

Name	Size (KB)	RR + RR*	RR*	
			cap.=1	cap.>1
Accel	39	18	0, 0	0, 0
Azurad	69	10	0, 0	0, 0
CableGrind	9891	989	988, 0	0, 0
CableGrindLlama	1071	101	101, 0	0, 0
Childs_Game	29	1	0, 0	0, 0
CML	37	9	0, 0	0, 0
cyber_blogger	43	12	0, 0	0, 0
Divelogger2	48	8	0, 0	0, 0
ECM_TCM_Simulator	56	9	0, 0	0, 0
Enslavednode_chat	25	5	0, 0	0, 0
EternalPass	19272	16382	0, 0	0, 0
expression_database	27	8	1, 0	0, 0
FileSys	47	72	0, 0	1, 1
Filesystem_Command_Shell	43	10	0, 0	0, 0
Finicky_File_Folder	33	2	0, 0	0, 0
Fortress	40	1	0, 0	0, 0
FUN	182	4	0, 0	0, 0
Grit	45	17	1, 0	0, 0
middleware_handshake	69	3	0, 0	0, 0
Mixology	153	4	0, 0	0, 0
Network_File_System_v3	72	5	0, 0	0, 0
On_Sale	59	100	0, 0	0, 0
online_job_application	24	28	0, 0	0, 0
online_job_application2	24	26	0, 0	1, 1
pizza_ordering_system	52	45	0, 0	1, 1
RAM_based_filesystem	34	9	0, 0	0, 0
reallystream	667	5	0, 0	0, 0
SCUBA_Dive_Logging	36	8	0, 0	0, 0
Terrible_Ticket_Tracker	42	10	0, 0	0, 0
TVS	16	3	0, 0	0, 0
vFilter	27	3	0, 0	0, 0
Virtual_Machine	46	12	0, 0	0, 0
virtual_pet	21	15	0, 0	0, 0
XStore	41	1	0, 0	0, 0

The last two columns show the number of RR gadgets with different capacities. The pairs of numbers indicate the count of RR* gadgets found through aggressive and conservative assessment, respectively. See Section 7.1 to review the definitions of aggressive and conservative assessment.*

952 KB. There are 14 of them written in C++ and the other 20 were written in C.

8.1.2. Open Source Software Binaries. Table 5 lists the open source software analyzed in the evaluation. Most of them have been used by previous research for evaluating the effectiveness of new code-reuse attacks and defenses. While it is known that applications like browsers and databases are commonly targeted by code-reuse attacks, the security of programming toolchains has also been a concern for decades [77]. A recent security incident showed that compilers compromised by malicious parties can cause devastating consequences [9]. All applications are compiled by GCC 4.8 for x86 Linux. The `objdump` disassembler from GNU Binutils is used to decode binaries into assembly. We compiled the open source software with "legacy settings" such that no relocation or debug information is kept in the exectuables.

8.2. Gadget Statistics

In the experimental assessment, we used the default code-pointer identification algorithm (review Section 7.1 for details) to locate RR and RR* gadgets in each analyzed binary, with both the aggressive and conservative assessment strategies. The last three columns in Table 4 are the numbers of randomization resilient gadgets found in the CGC binaries, while Table 6 summarizes the results for the real-world open source applications. Figure 6 additionally shows in each open source application binary the distribution of RR* gadgets classified by capacity.

TABLE 5. ANALYZED OPEN-SOURCE APPLICATIONS

Software	Version	Category	Binary	Size (MB)
Chromium	42.0	Browser	chrome	127.64
Firefox	36.0	Browser	libxul	72.67
MongoDB	3.0.5	Database	mongod	19.42
MySQL	5.7.8	Database	mysqld	21.92
Clang	3.7.0	Compiler	clang	51.43
GCC	5.2.0	Compiler	cc1plus	19.57
VirtualBox	5.0.6	Hypervisor	VirtualBox	10.05
QEMU	2.4.0	Emulator	qemu-aarch64	6.48
NGINX	1.8.1	Web Server	nginx	0.61

TABLE 6. STATISTICS OF USABLE GADGETS

Software	RR + RR*	RR*			
		Aggressive		Conservative	
		Count	Ratio	Count	Ratio
Chromium	131492	30229	23.0%	17824	13.6%
Firefox	98299	16569	16.9%	11689	11.9%
MongoDB	10978	2203	20.1%	1188	10.8%
MySQL	12308	2632	21.4%	901	7.4%
Clang	8972	1448	16.1%	753	8.4%
GCC	5952	198	3.3%	51	0.9%
VirtualBox	6600	464	7.0%	201	3.0%
QEMU	931	40	4.3%	0	0.0%
NGINX	30	9	30.0%	4	13.3%

See Section 7.1 to review the definitions of aggressive and conservative assessment.

In general, RR and RR* gadgets are widely available in the majority of the analyzed binaries, even with the conservative assessment strategy adopted. Indeed, since the CGC binaries are much smaller in size than the real-world applications, they contain much fewer usable gadgets. *However, even such few gadgets could lead to successful security beaches.* We show this later in Section 8.3.

For the open source applications, we further inspected the sensitive functions invoked by the randomization-resilient gadgets. Table 7 gives the number of RR and RR* gadgets calling sensitive functions in each assessed binary. In this evaluation, we only took low-level library functions into account, including standard libc function, POSIX APIs and their wrappers provided by third-party utility libraries. These functions are grouped into six classes. It can be seen that although the distributions of sensitive functions in different applications are diverse, many categories of sensitive functions are widely available through RR and RR* gadgets. We did not perform the sensitive function inspection for CGC binaries, since they were originally developed for the DECREE system where only seven system calls are available. Moreover, the CGC binaries are mostly proof-of-concept applications for testing security analysis tools and the majority of them rarely utilize any system-level functionality.

8.3. Case Studies

To demonstrate that the reported results can help software maintainers understand the severity of potential risks,

TABLE 7. NUMBER OF RR AND RR* GADGETS CALLING SENSITIVE FUNCTIONS GROUPED BY FUNCTIONALITY

	Process Management (e.g., execv, fork)	Memory Management (e.g., malloc, memcpy)	String Utility (e.g., strcat, stpcpy)	File (e.g., fopen)	Network (e.g., send, recv)	Library Loading (e.g., dlopen, dlsym)
Chromium	2227	1635	342	221	43	3
Firefox	154	11176	178	438	35	19
MongoDB	207	2171	37	16	10	0
MySQL	591	3671	366	16	1	0
Clang	7	2967	687	4	0	0
GCC	19	84	40	49	0	0
VirtualBox	0	2384	1	0	0	0
QEMU	1	77	2	18	11	0
NGINX	0	3	0	1	5	0

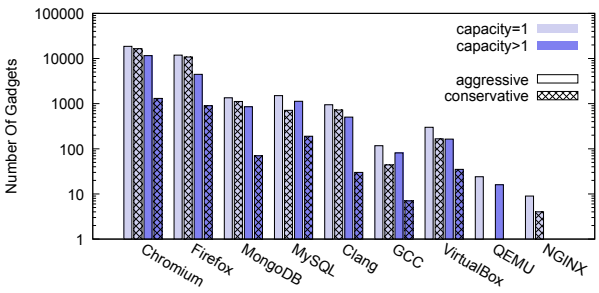

Figure 6. Distribution of RR* gadgets with trivial and non-trivial capacities. See Section 7.1 for definitions of aggressive and conservative assessment.

we undertook several case studies on the assessed binaries to investigate if they are indeed exploitable within our threat model. We studied 27 of the 34 CGC binaries. For the open source applications, we studied Chromium, GCC, and NGINX.

8.3.1. CGC Binaries. All CGC binaries are shipped with proofs of vulnerabilities (PoV) that demonstrate how to exploit the planted software flaws. There are two types of PoVs. The type I PoVs can hijack the program counter (the `eip` register) and control two general purpose registers. The type II PoVs can achieve arbitrary memory read and write. Since code-reuse attacks are control flow hijacking attacks, type I PoVs are more suitable for initiating the exploitation. A total of 27 analyzed CGC binaries contain type I PoVs. Note that the PoVs only demonstrate the feasibility of initiating attacks but not that attackers can achieve meaningful goals after hijacking the control flows, so further studies are required. We have verified that exploiting these PoVs do not require the knowledge of binary layout, meaning they are still exploitable even if the binaries are randomized.

We considered two types of code-reuse exploits in this case study. The first is to misuse the core functionality of the attacked program, e.g., logging in without proper authentication and tampering with the integrity of in-memory data. The second type of exploits is to reach a system call with malicious arguments. Our case studies concluded that 15 of the 27 binaries are vulnerable to core functionality misuse attacks and another five binaries are vulnerable to system call misuse attacks. For the remaining seven binaries, the contained gadgets are not powerful or versatile enough to launch any meaningful exploits.

It may be surprising that there are so many exploitable binaries, considering that most of the binaries do not contain so many RR and RR* gadgets (see the last three columns in Table 4). In particular, the lack of RR* gadgets in some CGC binaries means that it may be difficult to chain the gadgets together. Nonetheless, we found that there are two cases where the CGC binaries can be exploited without enough RR* gadgets.

The first case is that many CGC binaries are interactive applications which periodically take user input. Every time the input is taken, the vulnerability can be triggered. As a consequence, the attacker can achieve its goal with multiple rounds of attacks instead of constructing a single lengthy gadget chain. In the second case, the binaries are vulnerable to single-gadget attacks. In the `On_Sale` binary, for example, functions used to update product prices happens to be valid RR gadgets. Furthermore, the two buffer overflow vulnerabilities in `On_Sale` allows us to control a function pointer and its parameters. As such, we can secretly update the price of any product to our convenience.

8.3.2. Chromium. The assessment shows that Chromium contains a large number of usable gadgets, with many of them possessing strong capabilities of assessing and modifying system states. This strongly indicates that Chromium is not an appropriate target for rewriting-based randomization. To justify this statement, we constructed a proof-of-concept exploit for Chromium.

To bootstrap the attack, we reintroduced a previously discovered but currently fixed vulnerability, reported as CVE-2014-3176 [6], into the inspected version of Chromium. Starting with this vulnerability, we are able to implement an exploit that complies with our threat model, with which we can read (or write) an arbitrary file that the attacked browser has access to.

The exploit is triggered by navigating the vulnerable browser to an HTML page with malicious JavaScript code embedded. CVE-2014-3176 is a buffer overflow vulnerability that allows attackers to create a JavaScript array whose recorded length is larger than its allocated storage. Therefore, if another JavaScript object is allocated right after the corrupted array, attackers will be able to manipulate the meta data of that object and further trigger various unintended behavior [7]. We define a JavaScript

6. CVE-2014-3176 has been used to build proof-of-concept exploits by previous work [34]. We choose this vulnerability because it is one of the most well documented.

function (`JSFunction`) object and store the reference to this object into the overflowed part of the malformed array, such that we can leak the address of this function object and overwrite its pointer with the address of our first gadget. We can then initiate the attack by calling the function object in JavaScript with the payload and pointers to the payload as arguments.

We reused three RR* gadgets and two RR gadgets to build the attack chain. Each gadget is called exactly once. One of the RR* gadgets has the capacity of two while the others have the capacity of one. The attack sequence can be briefly described as follows. The 2-capacity RR* gadget serves as the first gadget and it calls the one of the 1-capacity RR* gadgets, which prepares necessary arguments for the first RR gadget and calls it. The first RR gadget opens a file whose name is provided by attack payload. The control flow then returns to the 2-capacity RR* gadget, which further calls another 1-capacity RR* gadget with its second steerable call site. The called 1-capacity RR* gadget invokes the second RR gadget with arguments loaded from attack input. The second RR gadget, which is the last hop of the attack, reads (or writes) the previously opened file.

8.3.3. GCC. Many of the real-world applications we inspected have at least hundreds of usable gadgets, and it is not surprising that they could be vulnerable to code-reuse attacks. On the other hand, it is unclear whether we can construct attacks with a shallow gadget pool. To investigate this matter, we particularly studied the `cc1plus` binary from GCC which has only 51 RR* gadgets when the conservative assessment method is applied. We managed to build a proof-of-concept exploit with which attackers can open an arbitrary file in the system hosting the vulnerable `cc1plus` binary and writes almost arbitrary textual content into that file, as long as the attacked process is granted such permissions.

To initiate the attack, we follow an approach adopted by previous work [68], i.e., injecting artificial memory vulnerabilities into the software. The attack employs three RR* gadgets and three RR gadgets, with a much more complicated chaining process compared to the Chromium exploit. In particular, we find it challenging to invoke the required system functions. Although the research of COOP has proposed several solutions to this problem, we cannot find the corresponding gadget patterns described by COOP.

8.3.4. NGINX. Only 30 usable gadgets are discovered in NGINX even with the aggressive strategy applied. With this small amount of potential gadgets, we were able to afford a thorough study by manually inspecting all of them. After the investigation, we believe that an attack fitting our threat model is not possible. Therefore, NGINX is unlikely to be vulnerable with the protection of rewriting-based fine-grained randomization.

Since an earlier version (1.7.11) of NGINX has been shown vulnerable to the "Control Jujustu" attack [43], we particularly examined the two functions employed by that exploit as gadgets. We found that although the addresses of those functions are taken in the source code, they are never stored in binary data due to compiler optimization, making the attack unfeasible within our threat model. The version difference is insignificant regarding this observation.

The case study on NGINX suggests that although code randomization for legacy binaries is generally vulnerable, the vulnerability may not always manifest. It is still reasonable to consider adopting the defense in certain scenarios as long as the potential risks have been carefully assessed and confirmed to be negligible.

9. Discussion

9.1. Implication of Assessment Results

For binaries that carry a notably large number of usable gadgets identified by the assessment, users should consider employing additional defenses to patch the security holes left by rewriting-based randomization. No matter how many gadgets the randomization manages to eliminate in the retrofitted binary compared with an unprotected copy, attackers may still have more than enough gadgets to construct severe exploits. In such situations, if users do have access to the source code, they should definitely consider adopting randomization techniques not relying on legacy binary rewriting. Otherwise, users will have to deploy defenses other than fine-grained randomization, even at a higher cost.

In case the assessment reports that only a relatively small number of gadgets are still usable after the binary is rewritten and randomized, the best option for users of this binary is probably to ask professional security analysts to inspect these usable gadgets and investigate if it is possible to construct code-reuse attacks that can pose threats to the production system. In such case, our risk assessment can significantly narrow the scope of gadgets to manually inspect and can provide helpful information about gadget capability, and thus significantly reduce the workload of the security analysts.

9.2. Potential Mitigations

Without relocation information, there are two potential directions for developing mitigations for the vulnerability caused by run-time address translation. The first is to limit attackers' access to address translation mechanisms while the second is to develop new legacy binary rewriting techniques without employing run-time address translation. For the sake of improving binary randomization, the second direction may be more meaningful. The reason is that to follow the first direction, the retrofitted binaries need to distinguish normal and malicious executions. However, this ability, e.g., a more fine-grained CFI technique, makes randomization lose its significance.

Although it is possible to improve binary randomization through compiler assistance such that the rewriter can rely on additional information [35], [36], the cost would be the ability to protect legacy and proprietary binaries. There are also attempts to cutting off code-reuse attack initiation steps by eliminating memory vulnerabilities at the first place, but this method is not specific to the scenario which our assessment focuses on. Moreover, it is generally believed that eliminating all possibilities of memory corruption is not practical even for newly developed software [76].

To the best of our knowledge, there is unlikely an instant mitigation to the problem revealed by our research at this point. The resolution of this problem requires long-term effort from the research community, especially breakthroughs in binary static analysis. Before that, our assessment will remain meaningful and necessary.

10. Related Work

10.1. Code Randomization

One of the most widely deployed code randomization practices is the coarse-grained ASLR [10] adopted by most modern operating systems. It was shown that ASLR can be defeated by brute force on 32-bit platforms [72] and is extremely vulnerable to memory disclosures. Also, ASLR requires support from the program loader. If the program loader itself is buggy, the effect of ASLR can be easily nullified [41].

Since coarse-grained randomization alone becomes ineffective to mitigate code-reuse exploitation, defenders started to improve randomization granularity. At first, fine-grained randomization was achieved by transforming the source code. Bhatkar et al. [19] proposed a source-to-source transformation to randomize program address space at the function level. Later development of similar techniques can achieve code randomization at compile time [36], link time [45], load time [53], [15], and even run time for just-in-time compiled code [49].

To protect legacy and proprietary software, researchers combined fine-grained randomization with binary rewriting. The binary stirring technique introduced by Wartell et al. [80] can randomize stripped binaries at the basic block level. Binary stirring was later adopted by O-CFI [61], a defensive technique that explicitly combines fine-grained code randomization and coarse-grained CFI. Another fine-grained randomization technique called ILR [48] implemented the address-translation mechanism with a per-process virtual machine trapping all indirect control-flow transfers and redirecting the targets to randomized addresses. Pappas et al. [63] proposed in-place code randomization, transforming binary code to a different form without moving basic blocks. Based on in-place randomization, Isomeron [38] and CodeArmor [30] further introduce probabilistic security at program run time.

10.2. Code-Reuse Attacks.

The idea of reusing the code of victim binaries can be traced back to the return-to-libc attack [40] which reuses a single function providing sensitive functionality. A systematic introduction to Turing-complete return-oriented programming was given by Shacham [71]. Since then, the scheme of ROP attacks has been specialized to be more effective in certain scenarios. For instance, Göktaş et al. [47] presented an ROP attack using gadgets of much various lengths so that they break the conventional assumption on gadget size, which is the foundation of some ROP defenses [64], [32]. Schwartz et al. [69] developed a code-reuse attack "compiler" to automate the construction of ROP attacks. Jump-oriented programming [28], [22] was proposed later in response to defenses specific to ROP, such as return-less binary [57]. Carlini and Wagner [27] invented history-hiding ROP attacks to evade abnormal branch detection.

Many of the attack techniques proposed more recently are combined with new insights. Götas et al. [46] demonstrated an attack which combines COP and ROP, resilient to coarse-grained CFI without a shadow stack. Carlini et al. [26] introduced control-flow bending, a code-reuse technique with the capability of bypassing a most strict fine-grained CFI implementation. Later shown by Evans et al. [43] and Conti et al. [34], even fine-grained CFI implemented with shadow stacks could be vulnerable to code reuse in certain cases. Counterfeit object-oriented programming [68] is able to bypass coarse-grained CFI and code-pointer separation [54] on C++ binaries that contain special virtual function invocation patterns.

The just-in-time (JIT) code reuse was invented to breach fine-grained code randomization, by searching for gadgets in the victim program's address space and construct gadget chains on the fly [74]. JIT code reuse requires the presence of memory disclosure vulnerabilities or side channels [70] to first disclose the executable memory of the attacked programs. This method was later extended from attacking scripting environments to exploiting vulnerable network services that automatically restart after crashes [21]. Data-oriented attack [29], [50], [51] is a code-reuse scheme without gadget programming. By manipulating non-control data, data-oriented attacks manage to feed malicious arguments to sensitive functions through perfectly valid control flows.

11. Conclusion and Call for Actions

We showed that current code randomization techniques based on legacy binary rewriting are still vulnerable to code-reuse attacks. With an in-depth analysis on this vulnerability, we proposed a systematic approach to assessing the potential risks of deploying such insecure defenses. We have assessed various binaries and demonstrated that the exposed security defect is indeed exploitable in certain cases. Through this work, we deliver three implications and calls for actions:

- Universally effective protection for legacy binaries is generally unfeasible at this point.
- For a particular binary, randomization is still a reliable security countermeasure if the risk assessment finds very few randomization-resilient gadgets.
- In case there are a large number of gadgets detected, users should consider more heavy-weight protections, e.g., fine-grained CFI and reassemble disassembling, even if that leads to considerable engineering cost or run-time overhead.

We hope our work can shed light on future research regarding binary retrofitting and fine-grained randomization.

Acknowledgements

We thank the reviewers for their valuable feedback. The work was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2912, N00014-16-1-2265, and N00014-17-1-2894.

References

[1] "Cyber fault-tolerant attack recovery (cfar)," http://www.darpa.mil/program/cyber-fault-tolerant-attack-recovery.

[2] "Cyber grand challenge," https://www.darpa.mil/program/cyber-grand-challenge.

[3] "Darpa challenges sets for linux, windows, and macos," https://github.com/trailofbits/cb-multios.

[4] "Exploiting misuse of Python's "pickle"," https://blog.nelhage.com/2011/03/exploiting-pickle/.

[5] "Galois awarded $10m DARPA contract to make legacy systems more secure," http://goo.gl/GSC63m.

[6] "Industrial control systems: What are the security challenges?" http://goo.gl/DF4wHt.

[7] "Issue 386988 - chromium," https://bugs.chromium.org/p/chromium/issues/detail?id=386988.

[8] "Legacy systems continue to have a place in the enterprise," http://goo.gl/mT9dfo.

[9] "Novel malware XcodeGhost modifies Xcode, infects Apple iOS apps and hits App Store," http://goo.gl/rYKeOF.

[10] "PaX address space layout randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt.

[11] "Study: US government spends $36 billion a year maintaining legacy systems," https://goo.gl/qnhjzu.

[12] "Windows 3.1 is still alive, and it just killed a french airport," https://news.vice.com/article/windows-31-is-still-alive-and-it-just-killed-a-french-airport.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, 2005, pp. 340–353.

[14] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

[15] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Security '14, Aug. 2014, pp. 433–447.

[16] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *Proceedings of the 21st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '00.

[17] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security '14, 2014, pp. 845–860.

[18] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proceedings of the 25th Network and Distributed Systems Security Symposium*, NDSS '18, 2018.

[19] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the 14th USENIX Security Symposium*, USENIX Security '05, Aug. 2005, pp. 255–270.

[20] 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, 2015, pp. 268–279.

[21] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, S&P '14, 2014, pp. 227–242.

[22] 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, 2011, pp. 30–40.

[23] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, vol. 4, no. 3, 2008.

[24] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, 2016.

[25] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, 2008, pp. 27–38.

[26] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium*, USENIX Security '15, 2015.

[27] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security '14, Aug. 2014, pp. 385–399.

[28] 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, 2010, pp. 559–572.

[29] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th USENIX Security Symposium*, USENIX Security '05, 2005, pp. 177–191.

[30] X. Chen, H. Bos, and C. Giuffrida, "CodeArmor: Virtualizing the code space to counter disclosure attacks," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, EuroS&P '17, 2017.

[31] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, CODASPY '17, 2016, pp. 50–61.

[32] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attack," in *Proceedings of the 21st Symposium on Network and Distributed System Security*, NDSS '14, 2014.

[33] M. Christodorescu and S. Jha, "Testing malware detectors," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, 2004, pp. 34–44.

[34] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS '15, 2015.

[35] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, 2015.

[36] 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 Conference on Computer and Communications Security*, CCS '15, pp. 243–255.

[37] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in PHP: Automated POP chain generation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014, pp. 42–53.

[38] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Proceedings of the 22nd Network and Distributed Systems Security Symposium*, NDSS '15, 2015.

[39] Z. Deng, X. Zhang, and D. Xu, "BISTRO: Binary Component Extraction and Embedding for Software Security Applications," in *Proceedings of the 18th European Symposium on Research in Computer Security*, ESORICS '13, 2013, pp. 200–218.

[40] S. Designer, "return-to-libc attack," *Bugtraq*, Aug 1997.

[41] A. Di Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "How the ELF ruined Christmas," in *Proceedings of the 24th USENIX Security Symposium*, USENIX Security '15.

[42] S. Esser, "Utilizing code reuse/ROP in PHP application exploits," *BlackHat USA*, 2010.

[43] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS '15, 2015.

[44] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, HotOS '97, May 1997, pp. 67–72.

[45] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Security Symposium*, USENIX Security '12, 2012, pp. 475–490.

[46] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S&P '14, 2014, pp. 575–589.

[47] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security '14, 2014, pp. 417–432.

[48] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proceedings of 33rd IEEE Symposium on Security and Privacy*, S&P '12, 2012, pp. 571–585.

[49] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: Transparent code randomization for just-in-time compilers," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, 2013, pp. 993–1004.

[50] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Proceedings of the 24th USENIX Security Symposium*, USENIX Security '15, 2015, pp. 177–192.

[51] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, S&P '16, 2016, pp. 969–986.

[52] K. Johnson and M. Miller, "Exploit mitigation improvements in windows 8," *Black hat USA*, 2012.

[53] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, 2006, pp. 339–348.

[54] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, Oct. 2014, pp. 147–163.

[55] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, S&P '14, May 2014, pp. 276–291.

[56] G. Laurence and C. Frohoff, "Marshalling pickles," AppSec California, 2015.

[57] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

[58] F. Lindner, "Cisco IOS router exploitation," *Black Hat USA*, 2009.

[59] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pp. 190–200.

[60] J. Ming, D. Xu, L. Wang, and D. Wu, "LOOP: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015, pp. 757–768.

[61] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Proceedings of the 22nd Symposium on Network and Distributed System Security*, NDSS, 2015.

[62] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[63] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, S&P '12, 2012, pp. 601–615.

[64] ——, "Transparent ROP exploit mitigation using indirect branch tracing," in *Presented as part of the 22nd USENIX Security Symposium*, USENIX Security '13, 2013, pp. 447–462.

[65] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '15, 2015, pp. 144–164.

[66] M. Ros and P. Sutton, "A post-compilation register reassignment technique for improving hamming distance code compression," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '05. ACM, 2005.

[67] M. E. Saleh, A. B. Mohamed, and A. A. Nabi, "Eigenviruses for metamorphic virus recognition," *Information Security, IET*, vol. 5, no. 4, pp. 191–198, 2011.

[68] 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 *Proceedings of The 36th IEEE Symposium on Security and Privacy*, S&P '15, 2015, pp. 745–762.

[69] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proceedings of the 20th USENIX Conference on Security*, USENIX Security '11, 2011, pp. 25–25.

[70] 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, 2014, pp. 54–65.

[71] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, 2007, pp. 552–561.

[72] 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, 2004, pp. 298–307.

[73] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu, "Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, 2012, pp. 309–318.

[74] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P '13, 2013.

[75] G. J. Sussman and G. L. Steele Jr, "Scheme: A interpreter for extended lambda calculus," *Higher-Order and Symbolic Computation*, vol. 11, no. 4, pp. 405–439, 1998.

[76] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S&P '13, 2013, pp. 48–62.

[77] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984.

[78] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," 2017.

[79] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the 24th USENIX Security Symposium*, USENIX Security '15, Aug. 2015.

[80] 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 19th ACM Conference on Computer and Communications Security*, CCS '12, 2012, pp. 157–168.

[81] ——, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, 2012, pp. 299–308.

[82] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization." in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, 2016, pp. 367–382.

[83] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S&P '13, 2013.

[84] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, 2014, pp. 129–140.

[85] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 22nd USENIX Security Symposium*, USENIX Security '13, 2013, pp. 337–352.