An In-depth Analysis of the Code-Reuse Gadgets Introduced by Software Obfuscation

Naiqian Zhang, Zheyun Feng, and Dongpeng Xu

University of New Hampshire, Durham NH 03824, USA
{Naiqian.Zhang, Zheyun.Feng, Dongpeng.Xu}@unh.edu

Abstract. Software obfuscation techniques are commonly employed to resist malicious reverse engineering. However, recent studies indicate that obfuscation introduces potential vulnerabilities susceptible to code-reuse attacks because the number of code-reuse gadgets in obfuscated programs significantly increases. Understanding how different obfuscation techniques contribute to the emergence of these code-reuse gadgets is crucial for developing secure obfuscation schemes that minimize the risk of code-reuse attacks, but no existing study has investigated this problem.

To address this knowledge gap, we present a comprehensive study on the impact of software obfuscation on code-reuse gadgets in programs. Firstly, we collect and analyze metrics data of gadgets obtained from a benchmark of programs obfuscated using various techniques. By examining the statistical results, we establish quantitative and qualitative relationships between each obfuscation technique and the resulting gadgets. Our key findings reveal how obfuscation techniques introduce significant code-reuse attack risks to a gadget set from different measurement schemes. Secondly, we delve into the underlying mechanisms of each obfuscation technique and elucidate why they contribute to generating specific types of gadgets. Lastly, we propose a mitigation strategy that combines low-risk obfuscation methods. Evaluation results demonstrate that our mitigation strategy effectively reduces the risks associated with code-reuse attacks without compromising obfuscation strength.

Keywords: Software Obfuscation · Code-reuse Attack · Gadget

1 Introduction

Software obfuscation has become increasingly important in defending against malicious reverse engineering, with various obfuscation methods being designed and implemented in both academic prototypes and industrial tools [12, 20, 24, 25, 33]. Despite their widespread usage, the security aspects of these obfuscation techniques have received limited attention. One significant risk arises from the insertion of opaque code by obfuscators, which is often treated as a black box due to its lack of comprehensibility from the users' perspective. Previous research has shown that obfuscation can increase the number of gadgets in obfuscated

binaries [18, 19]. However, the underlying mechanisms of obfuscation and the reasons behind the surge of these gadgets have not been extensively explored.

In this paper, we conduct an in-depth examination of selected popular obfuscation methods and their impacts on introducing code-reuse gadgets. We first apply various obfuscation techniques to a program benchmark and measure different characteristics of the code-reuse gadgets within the obfuscated programs. To compare, we focus on three aspects: the code-reuse gadget set's quantity, type, and risk. We assign scores to each obfuscation method based on these metrics and generate a prioritized list. Consequently, we propose a mitigation strategy that combines one low-risk obfuscation method with another for the protected programs. Through evaluation, we demonstrate that this strategy significantly reduces the number of exploitable code-reuse gadgets while maintaining the same level of obfuscation complexity.

In our study, we obfuscate 900 programs from an obfuscation benchmark [3] with four well-known obfuscators in academia and industry, namely Tigress [12], Obfuscator LLVM [20], VMProtect [33] and Code Virtualizer [24]. These obfuscators collectively implement a wide range of prevalent obfuscation methods. Each program is built with a unique obfuscation configuration to facilitate our incremental analysis. By comparing gadget metrics between the unobfuscated programs and obfuscated programs employing a specific obfuscation technique, we gain insights into the inner mechanisms of each obfuscation method and their impacts on code-reuse gadgets. Our findings reveal that different obfuscation techniques pose varying levels of code-reuse attack risks to the original program. To summarize, our contributions are as follows:

- First, we conduct a systematic study that sheds light on how obfuscation techniques introduce code-reuse gadgets. Our study employs a combined measurement scheme encompassing quantitative, qualitative, exploitable metrics and code-reuse attack risk assessment.
- Second, we conduct an in-depth analysis of each obfuscation method, unveiling the key factors that influence the presence of code-reuse gadgets and gadget sets. We develop a comprehensive assessment mechanism that ranks the obfuscation methods based on their code-reuse risks.
- Third, we propose a mitigation strategy to minimize the risk of code-reuse attacks without compromising the complexity and strength of obfuscation. Evaluation results demonstrate that employing low-risk obfuscation techniques, or multiple instances of them on the original program, reduces the code-reuse attack risk compared to high-risk obfuscation methods, all while preserving the complexity of obfuscation.

2 Background

To provide a better understanding of our work, we begin by introducing the background of code obfuscation techniques and the fundamentals of code-reuse attacks.

2.1 Code Obfuscation

Code obfuscation involves transforming a normal program into a semantically equivalent but more complex form. This transformation makes it challenging to comprehend the obfuscated code, and as a result, obfuscation techniques are widely employed to protect proprietary code from reverse analysis by hackers. Popular obfuscation tools, such as Tigress, Obfuscator-LLVM, VMProtect, and Code Virtualizer, incorporate a range of obfuscation schemes, as shown in Table 1.

Table 1: Obfuscation schemes in popular obfuscation tools.

Type	**Description**
Control Flow Flattening	Transform a program's control flow into a flat dispatch structure inside a loop, where a variable decides the program's next step [22]. The code inside the loop is in a linear style without any branches.
Instruction Substitution	Replace one instruction with a more complex but equivalent form, which may bring additional instructions to perform intermediate steps. For example, $x \mid y \Rightarrow (x \wedge y) \mid (x \oplus y)$
Bogus Control Flow	Insert dummy path conditions without changing the original program semantics. Usually, the dummy branch is randomly filled up with garbage codes.
Virtualization	Create a custom virtual machine (VM) and then translate the original program into the VM's bytecode, so the program's behaviors hide in the complicated VM execution. Virtualization has been recognized as one of the most complex obfuscation methods [23, 26].
Just-In-Time Dynamic	Translate the program into a sequence of customized intermediate representative instructions. This new code part will be dynamically compiled into machine code at run-time.
Self-Modification	Insert special code patterns into the program, which can change other parts with the same functionality of the program during run-time.
Encode Components	Replace integers, integer variables, integer arithmetic, and string literal to more complicated and complex expressions and opaque representations. It looks similar to Instruction Substitution but has a wider range of action objects.

2.2 Code-Reuse Attack

In recent years, code-reuse attacks have emerged as a highly dangerous attacking technique [9]. In such attacks, attackers search for short code snippets, known as gadgets, within a normal program, which are combined to achieve malicious objectives. This technique originated from traditional return-to-libc attacks. Shacham demonstrated that a gadget set used in code-reuse attacks is Turing-complete, which is theoretically capable of performing any malicious behavior [31]. Subsequent research has further extended code-reuse attacks from various perspectives. For instance, gadgets can involve complex control flow structures [4], and dispatch gadgets [16, 17], multiple architectures [8, 13], call-preceded [9], and jump-preceded [11] gadgets have been introduced.

Practical code-reuse attacks typically aim to gain control of the victim's machine (root) or tamper the permissions of specific files. Table 2 lists commonly triggered system calls during malicious activities. Furthermore, exploiting code-reuse attacks necessitates the presence of at least one known memory write vulnerability in the victim program, allowing the attacker to write the payload to the stack. Attackers leverage these vulnerabilities as starting points for code-reuse attacks, which can exist in the original, obfuscated, or library code. Several existing tools [2, 5, 6, 10, 15, 27, 34] aid in the identification of these memory vulnerabilities. However, the focus of this work does not include the process of locating these memory vulnerabilities.

Table 2: The system calls commonly used in code-reuse attacks.

Syscall	**Description**
`execve`	Trigger a shell-like `/bin/bash` on the victim machine.
`mmap` `mremap`	Map a file controlled by attackers as executable and then redirect the execution to that tampered file.
`mprotect`	Mark a page that includes content controlled by an attacker as executable and then redirects the program counter toward that tampered page.
`fchown` `fchmod`	Change permissions of a file.

3 Code-Reuse Gadgets Introduced by Obfuscation

Characterizing the impact of obfuscation on code-reuse attack gadget sets presents a considerable challenge. While modern obfuscation methods introduce a large number of gadgets into the code-reuse attack gadget sets, there is a lack of prior research that offers precise analysis and conclusions in this field. Therefore, a detailed investigation of the gadgets introduced by obfuscation is necessary.

In this section, we thoroughly examine the listed obfuscation methods and analyze the principles and implementation details behind each. Subsequently, we apply these methods to the programs within an obfuscation benchmark, generating code-reuse gadget sets for each program. We then observe and compare the number and types of gadgets in each set before and after applying the obfuscation method. Furthermore, we conducted in-depth research on existing works and discovered that they primarily focused on analyzing the number of gadgets in the gadget set resulting from obfuscation. However, to better reflect the true potential risk, combining this analysis with qualitative assessments of the gadget sets and the searching strategies employed by existing code-reuse generation tools is essential. To provide a comprehensive measurement of the impact of obfuscation, we implemented a standardized measurement system that examines the code-reuse gadget sets before and after obfuscation. This system analyzes the gadget sets from multiple perspectives, including quantitative assessment, qualitative assessment, and identification of exploitable gadgets. This comprehensive analysis enables us to assess whether a gadget set carries a higher risk of code-reuse attacks.

3.1 Benchmark and Obfuscation Selection

We carefully selected 100 C programs from an obfuscation benchmark [3], ensuring diversity in program size, complexity, and functionality. When choosing the benchmark and programs, we considered the aspects of Ground Truth and Applicability. This benchmark encompasses a wide range of C programs and includes scripts that allow us to obfuscate the programs using our selected obfuscators. Notably, the "basic algorithm" and "small programs" sets within the benchmark consist of simple and basic programs that align well with the ground truth. Hence, we utilize them as the benchmark for our analysis.

We employed four popular obfuscation tools mentioned in the previous section to conduct our study. We follow three criteria to pick the obfuscation variants to make the study comprehensive:

1. The variant is offered by at least one of the selected tools.
2. The variant can be successfully performed on all the benchmark programs without errors or run-time crashes.
3. The variant can transform any snippets inside the program rather than only specific ones.

We select seven obfuscation techniques introduced at Table 1 based on these criteria. For virtualization, we performed both the source and binary code obfuscation. We generated 900 distinct obfuscation variants for the benchmark programs by integrating each chosen technique from the selected obfuscators. Our analysis did not consider the programs as statically or dynamically linked libraries. Generally, attackers can utilize gadgets included in library code when mapping the program's memory address at runtime.

To examine the impact of each obfuscation method on code-reuse attack gadget sets, we compared the different obfuscation variants against the original binary without any obfuscation applied. We applied only one obfuscation method to the original program strictly adhering to the default configurations at a time and did not consider overlapping multiple obfuscation methods. For each original program and obfuscation variant, we scanned the gadget set of each binary and conducted a detailed analysis. We categorized all gadgets from each gadget set into two groups: useful and useless gadgets. The classification was based on whether existing code-reuse exploitation construction tools could utilize a gadget. Useful gadgets refer to those utilized by existing exploitation tools to form valid gadget chains for code-reuse attacks, while useless gadgets have never been incorporated into gadget chains by any existing exploitation tools.

3.2 Gadget Measurement

Increment Rate. This quantitative metric assesses how the number of gadgets increases as a result of different obfuscation methods. We identify and calculate the new gadgets introduced by obfuscation that are not in the original binary. A code-reuse gadget refers to a binary code sequence ending with a control-flow transfer instruction such as `ret`, `jmp`, `call`, `syscall`, etc. The `jmp` instruction can further be categorized into conditional and unconditional jumps. After obfuscation, we count the number of gadgets and calculate the rate of increase for each gadget set. It is important to note that gadgets which remain semantically unchanged but are relocated to a new memory address after obfuscation are not considered as increased.

Exploitability. To better assess the code-reuse attack risk of a gadget set, we introduce the exploitable metric, which measures whether a gadget can be considered useful or if it poses a code-reuse attack risk to the gadget set. This metric determines the number of gadgets within a gadget set that automated search tools can utilize. Generally, more exploitable gadgets in a gadget set indicate a greater risk of code-reuse attacks.

To investigate existing code-reuse exploitation tools as well as their implementation details and search efficacy, we categorize them based on different searching methods into three aspects, as shown in Table 3. To guarantee the comprehensiveness of our exploitable metric, we select the code-reuse attacks searching tools following three criteria:

1. The tool is publicly available and easily used in the original and obfuscated programs.
2. The tool can generate valid chains that can perform at least one type of attack shown in Table 2.
3. The tool can clearly show all gadgets in a human-readable format in the attack chains.

Table 3: Methods of searching code-reuse attacks and representative tools.

Method	**Description**
Pattern matching and hard-coded searching	ROPGadget [28] and Ropper [29] both apply this strategy. They search for a bunch of known gadget patterns and require hard-coded rules based on built-in exploitation templates to chain gadgets together.
Symbolic execution and exploration	Angrop [1] and ROPium [32] identifies gadgets via symbolic execution. They maintain an intermediate representation of gadgets, which matches the symbolic execution result with the pre- defined semantic rules of the gadgets and chains of those gadgets together based on the attacker's specifications.
Program Synthesis	As the state-of-the-art exploitation technique, SGC [30] synthesizes logical formulas to represent the gadget chains between the starting and ending program states. Then it uses an SMT solver to verify the gadget chain is feasible.

We considered these criteria on each type of searching method in Table 3 and selected the representative tools from each category: ROPGadget, Angrop, and SGC. Then, we conducted analysis on the programs within the obfuscation benchmark using these selected tools, examining the chaining results. By counting the number of gadgets comprising the gadget chains found by each existing tool within each unobfuscated program, we identified the types of gadgets contributing more effectively to the code-reuse exploitation process. Figure 1 illustrates representative gadget chains discovered by each type of existing automated search tool. We observed that many of these gadget chains included gadgets performing assignments, such as those with `pop` and `mov` instructions. Therefore, gadgets with these instructions are considered exploitable gadgets.

Expressivity and Quality. In measuring the quality and expressivity of a gadget set, we selected a method proposed by Brown et al. [7]. For gadget set expressivity, this method evaluates the power of gadget set expressivity based on three aspects: practical ROP exploits, ASLR-proof practical ROP exploits, and Turing completeness. At a specific level of expressivity, a gadget set must contain at least one gadget that fulfills the required computational criteria for each of these aspects. For example, achieving practical ROP exploits necessitates the presence of gadgets that assign targeted values to specific registers, store values to memory, and trigger system calls, among others.

Regarding gadget set quality, the metric from Brown et al. focuses on the functionality of each gadget. This qualitative measurement assesses whether a

```
Gadget 1: 0x483635:
mov qword ptr[rsi],rax
ret
Gadget 2: 0x4106fe:
pop rsi
ret
Gadget 3: 0x452f37:
pop rax
ret
Gadget 4: 0x447d19:
xor rax, rax
ret
Gadget 5: 0x478ca0:
add rax, 1
ret
Gadget 6: 0x401752:
pop rdi
ret
Gadget 7: 0x4106fe:
pop rsi
ret
Gadget 8: 0x40165f:
pop rdx
ret
Gadget 9: 0x401213:
syscall
```

(a) ROPGadget

```
Gadget 1: 0x4f235d:
pop rax
ret
Gadget 2: 0x628f79:
pop rcx
ret
Gadget 3: 0x4c02ab:
mov dword ptr[rcx-0x7f],rax
ret
Gadget 4: 0x4f235d:
pop rax
ret
Gadget 5: 0x628f79:
pop rcx
ret
Gadget 6: 0x4c02ab:
mov dword ptr[rcx-0x7f],rax
ret
Gadget 7: 0x5be23e:
pop rdi
ret
Gadget 8: 0x514f31:
pop rdx
ret
Gadget 9: 0x74f9ff:
pop rsi
ret
Gadget 10: 0x401213:
syscall
```

(b) Angrop

```
Gadget 1: 0x48b0c6:
pop rbx
pop rbp
pop r12
pop r13
ret
Gadget 2: 0x418f47:
mov rax,r12
pop rbx
pop rbp
pop r12
pop r13
ret
Gadget 3: 0x473cbe:
mov rdx,r12
mov rsi,rbp
mov rdi,rbx
call qword ptr[r13+0x38]
```

(c) SGC

Fig. 1: Gadget chains built by existing code-reuse chain searching tools.

gadget exhibits side effects, such as conditional branches, additional memory or register operations, or stack pointer-related manipulations, which can affect exploit construction. For example, consider the gadget `{add eax, 1; ret;}`; it contains no intermediate instructions and thus has no side effect. On the contrary, the gadgets `{add esi, ecx; xor eax, eax; mov dword ptr[edx],r si; ret;}` have side effects. The instruction `xor eax, eax;` overwrites the value in `eax`, impacting the result set up by the attacker. Thus, gadgets without side effects and with single functionality are easier to exploit.

For our work, we employed the Gadget Set Analyzer (GSA) [7], a state-of-the-art tool for measuring gadget set properties. GSA calculates the gadget set expressivity by inspecting the first instruction of each gadget to determine if it satisfies the computational criteria for any of the three aspects mentioned earlier. The expressivity is then expressed as the total number of satisfied classes for each aspect. If the expressivity of a gadget set increases in one or more aspects, it is considered a potentially risky outcome. Regarding gadget set quality, GSA assigns a score to each gadget based on the presence of intermediate instructions that introduce side effects. The average quality score of the entire gadget set is then computed. If the score of the transformed gadget set surpasses that of the original gadget set, it indicates a potentially risky outcome.

Overall Risk. Lastly, in our gadget set measurement, we combine the aforementioned three measurement standards to derive a summarized metric for assessing the code-reuse attack risk of a gadget set resulting from different obfuscation transformations. We propose a formula

$$Risk_{CRA} = \frac{N_{(Chain\_Related)}}{N(Gadgets)} + V_{Expressivity} + V_{Quality}$$

that considers several statistical values related to the measurement variables of a gadget set. The risk value of the code-reuse attack gadget set is defined as the sum of three components: the expressivity value (sum of all three aspects), the quality value, and the ratio of the number of exploitable gadgets to the total number of gadgets in the set. This formula enables us to measure the code-reuse attack risk introduced by an obfuscation method to the gadget set.

4 Study Results

4.1 Gadget Quantity

We observed a substantial increase in the number of gadgets after obfuscation, as depicted in Figure 2. A comparison of gadget sets between the original program and obfuscated programs of various transformation types revealed an average increase of approximately 43 times in the number of gadgets. ROPGadget was used to calculate the gadget count due to its superior gadget-searching capabilities among existing tools. This highlights the significant impact of obfuscation methods on gadget sets' quantity and composition, introducing numerous different kinds of exploitable gadgets into the original programs.

4.2 Gadget Exploitability

We employed existing exploitation tools, as mentioned in Section 3, to search for code-reuse gadget chains in both the original and obfuscated programs. Subsequently, we analyzed which gadgets were frequently used in the gadget chains. Notably, for specific obfuscation methods, there was an apparent increase in the number of commonly used gadgets in the chains. Successful code-reuse attacks often involve triggering system-level calls such as *execve*, *mprotect*, *fchown*, and *mmap*. Exploitation tools must find appropriate gadgets to assign parameter values for these system calls. For instance, assuming the attacker intends to call *execve* to spawn a shell, the x86-64 calling convention requires assigning the system call number `0x3b` to the `rax` register, followed by assigning values to `rsi`, `rdi` and `rdx` as the arguments of *execve*.

By running existing tools for gadget chain searching on our test set, we collected over 300 chains, most of which exhibited similar patterns frequently used in gadget compositions mentioned in Section 3. The common exploit objective was to trigger the `execve` system call and spawn a shell.

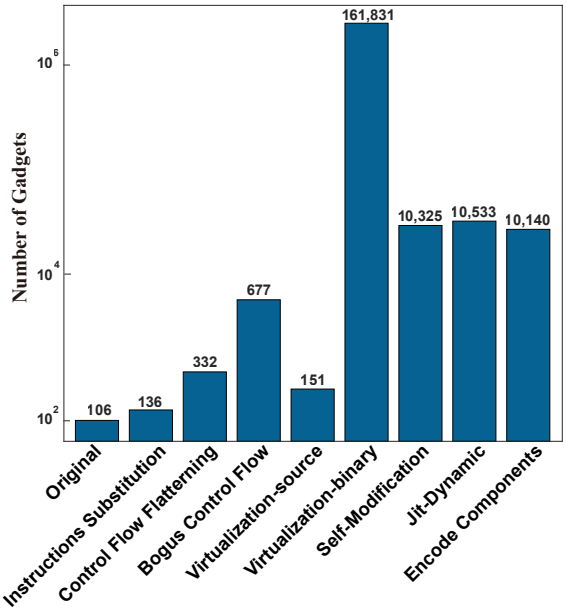


Fig. 2: Comparison of the number of gadgets from the original program and different types of obfuscated programs.

To gain a better understanding of the gadgets involved in the gadget chains, we tallied the number of exploitable gadgets in each gadget set and calculated their proportion within the set. This metric served as a crucial aspect for evaluating changes in gadget sets before and after obfuscation, as shown in Table 4. Most obfuscation methods led to an increase in the number of exploitable gadgets, indicating a worrisome sign for program protection against code-reuse attacks.

4.3 Gadget Quality

Our experiments revealed that specific obfuscation methods, particularly those involving opaque predicates or complex expression modifications like *Encode Components*, tended to increase the expressivity and quality value of a gadget set. The complete results are presented in the second and third columns of Table 5. Higher values in expressivity and quality indicate a greater range of gadget utility but an elevated risk of code-reuse attacks.

4.4 Code-Reuse Attack Risk

Based on the metric formula we defined for measuring the risk of code-reuse gadget sets and the results from our experiments, we ranked the risk value of each obfuscation method from low to high. A higher value indicates a greater risk posed by the respective obfuscation technique. The detailed risk values for each method are displayed in the fourth column of Table 5.

Table 4: The exploitable gadgets' number and the rates included in each gadget set of the original program and different obfuscated transformations. If the number of exploitable gadgets increases, this is considered a risk-increasing result.

Obfuscation	**#Exploitable Gadgets**	**Rates**
Original	36.25	30.5%
Instructions Substitution	41.47	31.4%
Control Flow Flattening	87.37	28.2%
Bogus Control Flow	101.25	16.1%
Virtualization-Source	58.45	38.9%
Virtualization-Binary	44625.36	27.6%
Self-Modification	2769.68	26.7%
Jit-Dynamic	2768.30	26.2%
Encode Components	2731.88	26.9%

Table 5: The second and third columns represent each gadget set's average expressivity and quality values for the original program and different obfuscated transformations. An obfuscation method that increases the expressivity value and decreases the quality value signifies an increase in risk. The fourth column presents the code-reuse attack risk value for each obfuscation method.

Obfuscation	**Expressivity**	**Quality**	**Risk Value**
Original	(4.125 / 6.125 / 2.175)	1.399	14.129
Instructions Substitution	(4.125 / 6.500 / 2.275)	1.322	14.536
Control Flow Flattening	(4.125 / 6.275 / 2.125)	1.263	14.070
Bogus Control Flow	(4.725 / 7.250 / 4.475)	1.068	17.679
Virtualization-Source	(4.950 / 8.450 / 3.025)	1.274	18.088
Virtualization-Binary	(44.950 / 78.450 / 53.025)	9.883	186.584
Self-Modification	(7.750 / 26.525 / 12.150)	1.786	48.460
Jit-Dynamic	(7.950 / 27.8 / 12.775)	1.798	50.585
Encode Components	(7.975 / 27.025 / 12.50)	1.809	49.578

5 The Anatomy of the Obfuscations and Gadgets

In light of the experimental results, this section offers an in-depth exploration and analysis of the varying gadget sets that correspond to each type of obfuscation method. We conducted a meticulous differential analysis of the binary code and the associated gadget sets, comparing singularly obfuscated variants against their original program counterparts. In addition, our investigation delves deeper into the implementation mechanisms of selected obfuscation techniques to comprehend the ways in which these techniques reshape the composition of gadget sets and the resulting impact on program security.

5.1 Instructions Substitution

Implementation Details. Instructions Substitution replaces binary operators with more complex sequences of instructions that have equivalent functionalities, such as arithmetic or Boolean operators. In Obfuscator-LLVM, this obfuscation technique supports integer operations including addition and subtraction, along with Boolean operators such as AND, OR, and XOR. For any given operator, there exist multiple equivalent expressions. The detailed implementation rules are shown in Table 6. The random selection of one of these equivalent expressions introduces a desirable diversity in the resulting binary instruction substitution. Moreover, instructions substitution significantly complicates the task of automatically searching for specific machine instruction patterns which are commonly used in symmetric ciphers such as XOR more difficult [20].

Table 6: The implementation rules of Instructions Substitution in Obfuscator-LLVM. X, Y, Z, and K are all integers.

Operator	Modified Equivalent Instructions

x = y + z	x = y - (-z)
	x = -(-y + (-z))
	x = y + k; x += z; x -= k;
	x = y - k; x += z; x += k;

x = y - z	x = y + (-z)
	x = y + k; x -= z; a -= k;
	x = y - k; x -= z; x += k;

x = y & z	x = (y ^!z) & y
x = y | z	x = (y & z) | (y ^z)
x = y ^z	x = (!y & z) | (y & !z)

Root Causes and Discussion. *Instructions Substitution* resulted in an increase of nearly 30% in new gadget generation compared to the original program. Those freshly introduced gadgets barely affected the number of exploitable gadgets but slightly increased the value of gadget quality and expressivity. We observed that a majority of the newly introduced gadgets are tied to arithmetic or logic operators. It introduces novel operators at the assembly level as it replaces one binary operator with a sequence of instructions. For instance, an obfuscated program that substitutes the expression `{x = y + z}` with `{x = -(-y + (-z))}`. This operation not only utilizes addition but also subtraction to accomplish the operation. As a result, a new gadget `{add al, 0x3d ; sbb al, 0x30 ; ret}` is added to the gadget set. Gadgets of this nature can significantly increase the expressivity of the gadget set, but they do not contribute to the generation of gadget chains.

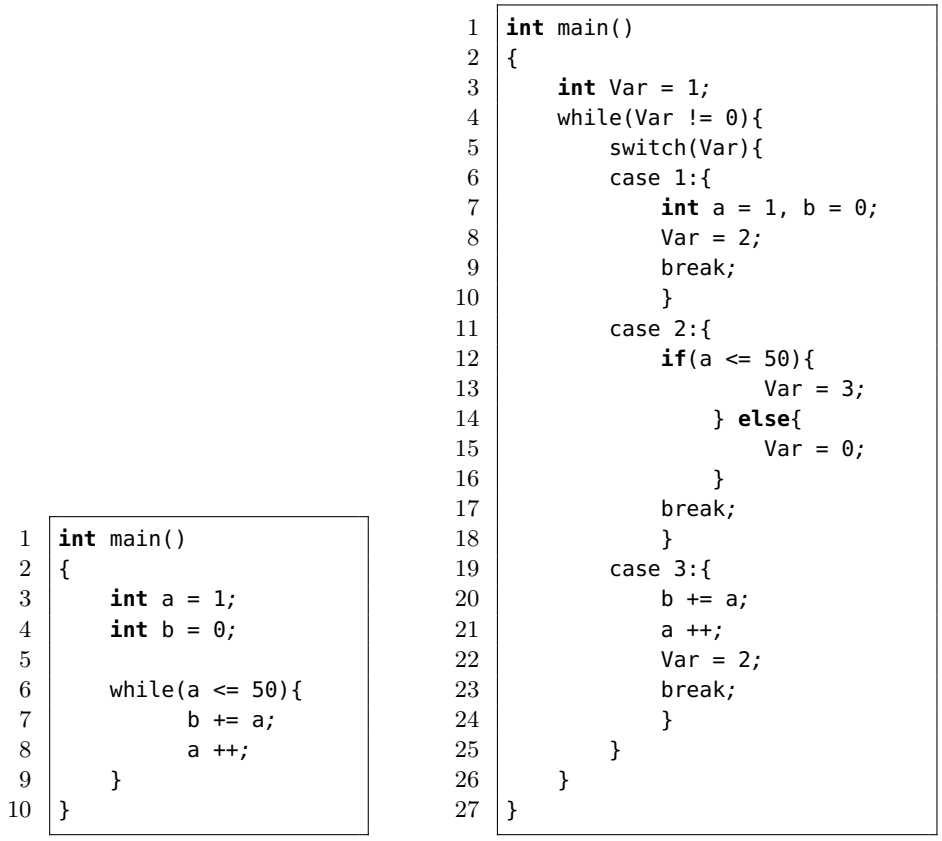
5.2 Control Flow Flattening

Implementation Details. Control Flow Flattening manipulated the principal structure of the source code into a format that conceals the targets of branches. Initially, each function was broken up into basic blocks. These blocks, regardless of their original nesting levels, are then arrayed in parallel within a `switch-case` statement. Each basic block resides in a distinct case structure, and the entire `switch` structure is encapsulated within a loop statement. The order of control flow is guaranteed by a control variable, which is assigned at the termination of each basic block as the predicates of the finishing the loop and selection of `switch` statement. Fig. 3b illustrates the obfuscated program that has applied the control flow flattening method to the original program shown in Fig. 3a. As can be observed, all basic blocks from the original program are at the same level in the obfuscated program, thus effectively concealing the loop structure of the original program.

Root Causes and Discussion. As demonstrated in Section 4, *Control Flow Flattening* contributed to a 213% increase in the introduction of new gadgets compared to the original program. Those newly introduced gadgets subtly impact the gadget set's exploitable gadgets, quality, and expressivity. The count of exploitable gadgets doubles, whereas the values of expressivity and quality remain unchanged or even decrease. We observed a substantial number of gadgets ending with a direct jump being introduced into the gadget set, these gadgets account for almost all of the newly introduced gadgets, even serving as the ending instructions for the exploitable gadget. The situation is directly related to the implementation mechanism of this transformation. *Control Flow Flattening* generally employs a `Switch-Case` structure to flatten an entire function's control flow graph. At the assembly level, a `Switch-Case` statement usually relies on a jump table and fills in the case names. It requires direct jumps with conditions to decide the control flow's direction. Consequently, the gadget set with this transformation introduces many gadgets ending with a direct jump. However, drawing upon our experience and corroborated by the state-of-the-art exploitation tools, we find that these newly introduced gadgets cannot be harnessed for constructing gadget chains for code-reuse attacks.

5.3 Bogus Control Flow

Implementation Details. *Bogus Control Flow* involves the insertion of spurious control flows within a function to reconstruct its corresponding control flow graph. The outcome is a chaotic control flow graph encompassing three irrelevant types of branches, all of which are shielded by opaque predicates. (1). The dead branch that is never engaged; (2). The superfluous branches that are invariably engaged; (3). The branches that are sporadically engaged. The first type involves the inclusion of a counterfeit block (which could be arbitrary code) within a basic block, giving the impression that it might be executed later, but in reality,



```
int main()
{
    int a = 1;
    int b = 0;

    while(a <= 50){
          b += a;
          a ++;
    }
}
```

(a)

```
int main()
{
    int Var = 1;
    while(Var != 0){
        switch(Var){
        case 1:{
            int a = 1, b = 0;
            Var = 2;
            break;
            }
        case 2:{
            if(a <= 50){
                    Var = 3;
                } else{
                    Var = 0;
                }
            break;
            }
        case 3:{
            b += a;
            a ++;
            Var = 2;
            break;
            }
        }
    }
}
```

(b)

Fig. 3: The sample programs before and after using Control Flow Flattening.

it is never executed. The second type involves the insertion of a true predicate midway through a basic block, creating the illusion that the original block is only intermittently executed. The third type involves the insertion of a variable predicate which occasionally directs the execution left or right, with the resulting paths being identical regardless of the direction chosen by the predicate.

Root Causes and Discussion. *Bogus Control Flow* resulted in a 538% increase in new gadgets compared to the original program. Those newly introduced gadgets augment the count of exploitable gadgets as well as the value of expressivity. However, most of those gadgets end with direct jumps, which cannot be used to generate a gadget chain and thus have no impact on the quality value. Furthermore, this transformation also brings tons of Control Flow Graph (CFG) nodes to the original CFG. These spurious CFG nodes necessitate a significant number of direct jumps with conditions to facilitate their integration into the original

CFG. Consequently, the gadgets introduced via *Bogus Control Flow* scarcely contribute to increasing the code-reuse attack risk.

5.4 Virtualization

Implementation Details. *Virtualization* involves the conversion of selected portions of code into bytecode, defined by a specialized virtual instruction set architecture. The bytecode is then emulated by an embedded interpreter on the actual machine during runtime. More specifically, the original code of a program is initially transformed into bytecode as per a custom virtual instruction set. Subsequently, the bytecode interpreter carries out execution following a Fetch-Decode-Dispatch procedure. The fetch step involves the retrieval of the next bytecode instruction, the decode step is responsible for decoding the instruction and its operands (if any), and the dispatch sets up the execution environment and calls the correct handlers.

Root Causes and Discussion. The *Virtualization* offered by Tigress led to the introduction of nearly 50% more new gadgets than the original program, meanwhile, the quality and the expressivity of the gadget sets remained unaltered. Given that Tigress implements the transformation at the source code level, we carefully examined the obfuscated source code and found that the bytecode and handlers are appended to the source code prior to compilation. Figure 4 provides an example of the bytecode in the obfuscated source file. Subsequently, a `switch-case` based dispatch structure is utilized to interpret the bytecode and map the bytecode to corresponding handlers. Aside from introducing a few gadgets with direct jumps following the dispatch process, this transformation in Tigress neither alters the control flow of the original program nor complicates the operation of individual instruction. As a result, no gadgets with practical functions are introduced.

For comparison, we also performed binary-level virtualization on the same benchmark using Code Virtualizer [24], a commercial software obfuscation product developed by Oreans Technologies. The binary-level virtualization brings tons of new gadgets, resulting in an increase of 1500 times more new gadgets than the original program. This tremendous increase is attributable to the binary-level virtualization embedding the entire virtual machine, its handler set, and the translated bytecode from the original code into the obfuscated program. Those components are equivalent to adding a complete virtual machine program to the original program, which greatly boosts the number of gadgets as well as the expressivity and quality values of the gadget sets. Therefore, binary-level virtualization poses greater code-reuse attack risks compared to source-code-level virtualization.

5.5 Just-In-Time Dynamic

Implementation Details. Tigress incorporated the *Just-In-Time* (JIT) dynamic techniques for obfuscation, which is implemented atop the MyJit library [21].

```
1 enum ops {
2   Return = 249, Store = 242, Formal = 183,
3   Plus = 178, Goto = 62, Load = 89, Local = 126
4 };
5
6 unsigned char bytecode[31]  = {
7   Formal,1,0,0,0,Load,Formal,0,0,0,0,Load,Plus,
8   Local,0,0,0,0,Store,Goto,4,0,0,0,Local,8,0,0,0,
9   Load,Return };
```

Fig. 4: The bytecode in Tigress-obfuscated source file.

This transformation converts a function F into a new function F' by integrating a sequence of intermediate code instructions. Upon execution of F', it dynamically compiles function F into machine code. Essentially, this technique generates machine code during run-time and then executes it. Figure 5 illustrates an example of an obfuscated program, which we utilize to describe the JIT dynamic procedure. Initially, the program constructs a new instance of the JIT compiler by invoking `jit_init()` on line 6. It then adds the intermediate code by calling `jit_add_op()`. Next, the JIT compiler translates the intermediate code into actual machine code by calling `jit_generation_code()`. Ultimately, the control flow is redirected to the code generated just now and the execution begins.

Root Causes and Discussion. *JIT Dynamic* results in a hundredfold increase in new gadgets compared to the original program, marking a substantial rise. The value of expressivity and quality also experience a significant rise. The implementation of JIT Dynamic id is dependent on a third-party library, with several functions in the library being called during the compilation and execution phases. This is equivalent to adding another new program into the original one, analogous to binary-level virtualization, thereby increasing the code-reuse attack surface of the original program. As a consequence, the gadget set contains a larger quantity of gadgets that can be used to construct gadget chains after transformation. Additionally, the gadget set exhibits higher expressivity and includes a greater number of gadgets with side effects.

5.6 Self-Modification

Implementation Details. *Self-Modification* aims to render functions self modifying during runtime. Typically, *Self-Modification* can be achieved by encrypting, encoding, or embedding certain parts of the code pattern into the original program, or by altering the program's execution path when it's running. Tigress amalgamates self-modification templates with two different types of transformations. One is the binary arithmetic expressions and comparisons, which inserts a binary code template at the top of the function and uses the template for modification. The other combines code virtualization and flattening, proving

```
1  int obf_target(int x, int y) {
2    ...
3    //First, initialization
4    static int (*_obf_target)(int x , int y) ;
5    int result;
6    p = jit_init();
7    jit_add_prolog(p, & _obf_target, 0);
8    jit_add_op();
9    jit_add_op();
10   ...
11   //Second, compilation
12   jit_generate_code(p);
13   ...
14   //Third, execution
15   result = (*_obf_target)(x, y);
16   return (result);
17 }
```

Fig. 5: JIT Dynamic implementation example in Tigress.

particularly effective after the introduction of indirect branches. Those indirect branches are transformed into other byte sequences that correspond to the direct jumps during runtime. This modification effectively thwarts deobfuscation methods that solely search for indirect branches, which have been removed from the original code.

Root Causes and Discussion. *Self-Modification* also results in an increase of new gadgets by a factor of 100 compared to the original program, with the value of quality and expressivity of the gadget set also increasing. This transformation inserts abundance of pre-defined code patterns into the obfuscated source code. Although these patterns are randomly employed during compilation, they remain attached, thereby enhancing the diversity of the original code and bringing more gadgets and higher risks.

5.7 Encode Components

Implementation Details. *Encode Components* comprises three components: encode literals, encode arithmetic, and encode data. The encode literals obfuscates integer literals (such as 100) and string literals (such as "100"), replacing them with opaque expressions or a function that is generated during runtime. The encode arithmetic substitutes integer arithmetic with more intricate and convoluted expressions based on certain fixed patterns. This means that for each operator, there are numerous possible encoding styles within this transformation, which are selected randomly. For example, figure 6 shows how an expression of integer addition can be replaced with a random Mixed Boolean-Arithmetic (MBA) expression of higher complexities, yielding the same arithmetic results.

```
1  int main(int x, int y) {
2      int x = 0;
3      int y = 5;
4      int z = x + y;
5      return 0;
6  }
```

(a) Before

```
1  int main(int x, int y) {
2      int x ;
3      int y ;
4      int z ;
5      x = 0;
6      y = 5;
7      z = ((x | y) << 1)
8          - (x ^ y);
9      return (0);
10 }
```

(b) After

Fig. 6: Encode Arithmetic Transformation.

The encode data targets integer variables, altering them to a non-standard data representation with the aim of concealing a variable's real value until it needs to be displayed. Moreover, if a variable is encoded, all variables associated with it will also be encoded. For instance, a random integer variable v can be replaced with $v' = \texttt{a} * \texttt{v} + \texttt{b}$. Figure 7 demonstrates the difference before and after this transformation. It can be observed that the real values of variable `x` and `z` are both obscured.

```
1  int main(int x, int y) {
2    int x = 5;
3    int z = x;
4    return 0;
5  }
```

(a) Before

```
1  int main(int x, int y) {
2   int x ;
3   int z ;
4   ...
5   x = 1583543192U;
6   z = (int )(1509654933U
7      * x - 2053070707U);
8   ...
9  }
```

(b) After

Fig. 7: Encode Data Transformation.

Root Causes and Discussion. *Encode Components* results in the introduction of new gadgets at a rate 100 times greater than the original program, also escalating the quality and expressivity values of the gadget set. This transformation is analogous to *Instructions Substitution* in general, but its implementation is more advanced. Our observation of the obfuscated source code revealed the inclusion of some Just-In-Time (JIT) techniques, indicating that the entire JIT library is attached to the obfuscated binary post-compilation. This explains why

Encode Components generates more chain-related gadgets and exhibits higher expressivity and quality values, even though it operates on the same transformation principle as *Instructions Substitution*.

6 Mitigation

In light of the code-reuse attack risk associated with each obfuscation scheme, we propose mitigation strategies in this section to counter-measure and minimize the risk without significantly compromising the obfuscation strengths. Our proposed solution is to limit the use of obfuscation schemes associated with high-risk values as much as possible while increasing the use of those with low-risk values. Additionally, to ensure the effectiveness of obfuscation while maintaining the complexity of obfuscated programs, we recommend repeated application of one low-risk obfuscation scheme or a combined use of multiple low-risk schemes.

6.1 Strategy

To this end, we designed a set of experiments to verify the correctness of our proposed solution. We first categorized the obfuscation schemes into two groups based on the risk values ranking obtained from Table 5. One group consists of low-risk value schemes: *Instructions Substitution*, *Control Flow Flattening*, *Bogus Control Flow*, *Virtualization(source)*; The other group includes high-risk value schemes: *Jit-Dynamic*, *Self-Modification*, *Encode Components*.

For the low-risk value group, we applied each obfuscation method to the same source program once, twice, and three times respectively, and then combined the three methods from Obfuscator-LLVM on the source program as another variant. We determined the number of times an obfuscation method was applied to the source code by adjusting the command-line parameters. For the high-risk value group, obfuscation methods were applied individually.

6.2 Evaluation

To gauge how effective our mitigation strategy restrains the growth of the code-reuse attack risk, we applied our metrics to the obfuscated programs shown in Table 7 column one. By calculating the risk values for each obfuscation variant, we evaluated the outcomes of our mitigation strategy.

The evaluation results revealed that our mitigation strategy is highly effective at diminishing the code-reuse attack risk. Applying low-risk obfuscation methods multiple times to the same original program can effectively curb the growth of the risk values of the gadget set compared to those high-risk methods, which typically have risk values nearing 50.

While focusing on the code-reuse attack risk value, we also assessed the effect of applying one obfuscation method multiple times. We used IDA Pro [14] to analyze the CFG of each variant binary based on the number of CFG nodes and verified whether the obfuscated variant could maintain the same obfuscation

Table 7: The gadget-set-related data for each type of obfuscation method with low-risk and high-risk values. The number after the method name indicates how often this method has been applied to the original program.

Obfuscation	**#Gadgets**	**#Exploitable**	**Expressivity**	**Quality**	**Risk Value**
Original Program	106	36.25	(4.125 / 6.125 / 2.175)	1.399	14.129
Instructions Substitution (1)	136	41.47	(4.125 / 6.500 / 2.275)	1.322	14.536
Instructions Substitution (2)	290	44.00	(4.125 / 6.500 / 2.275)	1.093	14.144
Instructions Substitution (3)	650	114.00	(4.750 / 14.500 / 7.500)	1.028	27.953
Control Flow Flattening (1)	332	87.37	(4.125 / 6.275 / 2.125)	1.263	14.070
Control Flow Flattening (2)	455	91.50	(4.500 / 8.525 / 3.000)	1.341	17.567
Control Flow Flattening (3)	460	87.50	(4.500 / 6.500 / 2.075)	1.269	14.534
Bogus Control Flow (1)	677	101.25	(4.725 / 7.250 / 4.475)	1.068	17.679
Bogus Control Flow (2)	2,469	229.21	(5.025 / 8.500 / 6.075)	0.946	20.638
Bogus Control Flow (3)	8,573	427.27	(5.125 / 15.500 / 7.275)	0.943	28.892
Virtualization-source	151	58.45	(4.950 / 8.450 / 3.025)	1.274	18.088
Virtualization-Binary	161,831	44,625.36	(44.950 / 78.450 / 53.025)	9.883	186.584
Jit-Dynamic	10,533	2768.30	(7.950 / 27.8 / 12.775)	1.798	50.585
Self-Modification	10,325	2769.68	(7.750 / 26.525 / 12.150)	1.786	48.460
Encode Components	10,140	2731.88	(7.975 / 27.025 / 12.50)	1.809	49.578

complexity. A program with more CFG nodes can be considered as one with higher obfuscation complexity. The results indicated that applying the low-risk obfuscation method (Control Flow Flattening used as an example here) multiple times does not diminish the complexity of the obfuscation results. The number of CFG nodes is 35, 37, and 36, respectively, corresponding to applying the obfuscation method once, twice, and three times. The original program only has 7 CFG nodes. Therefore, after multiple obfuscation iterations, the obfuscated program possesses more CFG nodes and a complex control flow, thereby better protecting it against reverse engineering. Meanwhile, the code-reuse attack risk values are 14.070, 17.567, and 14.534, respectively, corresponding to applying the obfuscation once, twice, and three times to the original program. These risk values are not as high as those of other high-risk obfuscation methods, indicating the efficacy of our mitigation strategy.

We also conducted a comparison between applying a high-risk obfuscation method once and applying a low-risk obfuscation method multiple times. We selected the *Jit-Dynamic*, which has the highest risk value among all methods as the representative of the high-risk value group, and *Bogus Control Flow* as the representative of the low-risk value group. The comparison results showed that, despite each CFG node of the Jit-Dynamic obfuscated program having numerous instructions, it only has three nodes on its CFG. On the contrary, a program obfuscated three times with the Bogus Control Flow method has over 1,000 CFG nodes. The low-risk obfuscation method evidently contributes more obfuscation complexity than the high-risk method. Moreover, using the low-risk obfuscation method multiple times results in a code-reuse attack risk value of 14.534, which is significantly less than the risk value of 50.585 associated with applying a high-risk obfuscation method once.

From the results of our evaluation, we noticed that our mitigation strategy not only curbs the growth trend of code-reuse attack risk on obfuscated programs but also significantly increases the complexity and intensity of them. Therefore, we conclude that when applying obfuscation techniques, it is preferable to choose a method with low-risk value and apply it multiple times to the original programs, while avoiding high-risk value methods.

7 Related Work

Concerning code-reuse attacks in obfuscated programs, our research demonstrated each obfuscation method's key factors that influence the presence of code-reuse gadgets and gadget sets. One recent work Gadget-Planner [35], also sheds light on code-reuse attacks on obfuscated code, but our work differs from it. Gadget-Planner simply compares the gadget chains before and after obfuscation and then focuses on building more complex attack chains from the obfuscated programs. However, the key point of our work is to investigate the underlying causes behind the variations in gadget chains introduced by different obfuscation methods. Our work fully exposes the attack risk by measuring the quantity, quality, and expressivity of gadget sets.

Comparatively, other similar works [18,19] focus solely on the number of gadgets within the gadget set, using the increase in gadget count to imply potential attack risks. Our analytical approach is more comprehensive and employs a reasonable risk metric, extending beyond a simple quantitative analysis. Our work ranks the risk levels associated with different obfuscation methods and reveals the root cause of each method. Additionally, we offer solutions to mitigate these risks, making it a complete and comprehensive research endeavor.

8 Conclusion

Software obfuscation techniques have become increasingly popular for protecting the logic of programs by introducing complex data and control flow structures that make the code difficult to comprehend. However, existing research predominantly focuses on cracking and reversing obfuscated programs, neglecting the potential security risks associated with these obfuscations. To address this gap, our study provides a comprehensive analysis of popular obfuscation techniques, specifically examining their impacts on code-reuse attack vulnerabilities. We have developed a measurement framework to assess the code-reuse attack risks introduced by different obfuscation methods. Our analysis reveals that each obfuscation method introduces varying levels of code-reuse attack risks, underscoring the need for a meticulous selection of obfuscation techniques. To mitigate these risks, we propose a mitigation strategy that combines low-risk obfuscation methods, effectively reducing the code-reuse attack vulnerabilities while maintaining strong obfuscation. In conclusion, our research highlights the

importance of considering the code-reuse attack risks associated with obfuscation techniques and provides valuable insights for developing secure obfuscation schemes. By adopting our proposed mitigation strategy, users can enhance the security of their software while maintaining robust obfuscation protection.

Acknowledgments. We would like to thank the anonymous reviewers and shepherd for their valuable feedback. This work was supported by NSF grants CNS-2022279 and CNS-2211905.

References

1. Angr-team: Angrop – a rop gadget finder and chain builder. https://github.com/angr/angrop (2021)
2. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Communications of the ACM (2014)
3. Banescu, S., Collberg, C., Pretschner, A.: Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. In: Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17) (2017)
4. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (AsiaCCS) (2011)
5. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (2017)
6. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. IEEE Transactions on Software Engineering (2017)
7. Brown, M.D., Pande, S.: Is less really more? towards better metrics for measuring security improvements realized through software debloating. In: 12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19) (2019)
8. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: 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)
9. Carlini, N., Wagner, D.: ROP is still dangerous: Breaking modern defenses. In: Proceedings of the 23rd USENIX Conference on Security Symposium (2014)
10. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: IEEE Symposium on Security and Privacy. IEEE (2012)
11. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (2010)
12. Collberg, C.: The Tigress C Obfuscator. https://tigress.wtf
13. Francillion, A., Castelluccia, C.: Code injection attacks on harvard-architecture devices. In: CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security. ACM (2008)
14. Hex-Rays: IDA Pro. https://www.hex-rays.com/products/ida/
15. Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of data-oriented exploits. In: 24th USENIX Security Symposium (USENIX Security 15) (2015)

16. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: IEEE Symposium on Security and Privacy (SP) (2016)
17. Ispoglou, K.K., AlBassam, B., Jaeger, T., Payer, M.: Block oriented programming: Automating data-only attacks. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2018)
18. Joshi, H.P., Dhanasekaran, A., Dutta, R.: Impact of software obfuscation on susceptibility to return-oriented programming attacks. In: 36th IEEE Sarnoff Symposium (2015)
19. Joshi, H.P., Dhanasekaran, A., Dutta, R.: Trading off a vulnerability: does software obfuscation increase the risk of ROP attacks. Journal of Cyber Security and Mobility (2015)
20. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM – software protection for the masses. In: Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO (2015)
21. Krajca, P.: MyJit Library. http://myjit.sourceforge.net/
22. László, T., Kiss, Á.: Obfuscating c++ programs via control flow flattening. Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica (2009)
23. Manikyam, R., McDonald, J.T., Mahoney, W.R., Andel, T.R., Russ, S.H.: Comparing the Effectiveness of Commercial Obfuscators Against MATE Attacks. In: Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW) (2016)
24. Oreans Technologies: Code Virtualizer: Total Obfuscation against Reverse Engineering. http://oreans.com/codevirtualizer.php
25. Oreans Technologies: Themida: Advanced Windows Software Protection System. https://www.oreans.com/themida.php
26. Polychronakis, M.: Reverse Engineering of Malware Emulators. (2011)
27. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: Application-aware evolutionary fuzzing. In: Network and Distributed Systems Security Symposium (2017)
28. Salwan, J.: Ropgadget (2011), http://shell-storm.org/project/ROPgadget/
29. Schirra, S.: Ropper (2019), https://scoding.de/ropper/
30. Schloegel, M., Blazytko, T., Basler, J., Hemmer, F., Holz, T.: Towards automating code-reuse attacks using synthesized gadget chains. In: European Symposium on Research in Computer Security (2021)
31. Shacham, H.: 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 (2007)
32. Souchet, A.: Ropium (2018), https://github.com/Boyan-MILANOV/ropium
33. VMProtect Software: VMProtect software protection. http://vmpsoft.com
34. Wang, Y., Zhang, C., Xiang, X., Zhao, Z., Li, W., Gong, X., Liu, B., Chen, K., Zou, W.: Revery: From proof-of-concept to exploitable. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (2018)
35. Zhang, N., Alden, D., Xu, D., Wang, S., Jaeger, T., Ruml, W.: No free lunch: On the increased code reuse attack surface of obfuscated programs. In: 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2023)