No Free Lunch: On the Increased Code Reuse Attack Surface of Obfuscated Programs

Naiqian Zhang*, Daroc Alden*, Dongpeng Xu*, Shuai Wang†, Trent Jaeger‡, Wheeler Ruml*
*University of New Hampshire
†Hong Kong University of Science and Technology
‡The Pennsylvania State University

***Abstract*—Obfuscation has been widely employed to protect software from the malicious reverse analysis. However, its security risks have not previously been studied in detail. For example, most obfuscation methods introduce large blocks of opaque code that are black boxes to normal users. In this paper, we show that, indeed, obfuscation can increase the attack risk. Existing gadget search tools, while able to find more gadgets in obfuscated code, do not succeed in assembling them into more exploits. However, these tools use strict pattern matching, greedy searching strategies, and only very simple gadgets. We develop Gadget-Planner, a more flexible approach to building code-reuse attacks that overcomes previous limitations via symbolic execution and automated planning. In a study across both benchmark and real-world programs, this approach finds many more exploit payloads on obfuscated programs, both in terms of number and diversity.**

***Index Terms*—Software Security, Code Obfuscation, Code-Reuse Attacks, Partial-Order Planning**

I. Introduction

In the last decade, software obfuscation has become a successful technique for protecting legitimate software against reverse engineering. Its fundamental methodology is to camouflage the real program behaviors with a huge amount of opaque code, which is generated and injected by the obfuscator. A variety of obfuscation methods, e.g., control flow flattening, opaque predicate, virtualization, have been widely implemented in academic and industrial tools [1]–[5]. Industry giants, e.g., Intel, DELL, and Siemens, are all well-known users of obfuscators [6].

While recent research from the security community focuses on the obfuscation and deobfuscation [7]–[11], the safety of obfuscation techniques themselves is little studied. All obfuscation tools inject tons of useless code into normal programs to hide the real program behavior [12]. For example, after Obfuscator LLVM obfuscation, the code size expands twice as large as the original program. Moreover, the users of these obfuscators have no knowledge about these newly injected codes, so users have to treat them as black-box and simply trust their safety. Previous work [13] has pointed out that a large number of programs crashed after obfuscation. Harshvardhan et al. [14], [15] find the number of Return-Oriented-Programming (ROP) gadgets increases in the obfuscated programs. This work hinted that the injected code from obfuscation might potentially increase the software attacking surface, but no research has been conducted to bridge this gap.

In this paper, we address this open problem by thoroughly studying the code-reuse attacks introduced by software obfuscation, and the final conclusion demonstrates that obfuscation indeed increases the attack risks. First, we apply popular obfuscators to a benchmark and then use existing code-reuse searching tools to find feasible payloads. The result shows that the number of gadgets increases significantly in the obfuscated programs.

However, a severe defect of existing code-reuse search tools is that they have very limited capability to build valid gadget chains from the obfuscated programs. Our further analysis reveals three reasons: restricted patterns, greedy searching strategies, and complex obfuscated gadgets. Consequently, they offer very limited help on studying the code-reuse attacks in obfuscated code.

To overcome this, we developed a new technique to resiliently search and build gadget chains from obfuscated programs. In outline, our method first symbolically analyzes each gadget candidate and pre-builds a pool of functional and diverse gadgets. Next, we use a search technique called automated planning to build gadget chains toward meaningful attack behavior. The gadgets are incrementally selected to address a need in the plan and infer ordering constraints over those gadgets based on their effects. In this way, our method comprehensively processes all types of gadgets without assuming any specific gadget patterns or searching strategies, overcoming the limitations of existing tools.

We have implemented the approach as a prototype called Gadget-Planner. Compared to peer tools, Gadget-Planner successfully constructs up to 30X more gadget chains from obfuscated program benchmarks. It also successfully finds 16 payloads on *netperf*, a real-world obfuscated program.

Contribution. In summary, our contributions are:

- We thoroughly inspect the risks of code-reuse attacks in obfuscated programs. The study result shows that obfuscation techniques introduce a huge number of gadgets; however, present automated tools cannot effectively construct code-reuse attack chains. We find that the restricted gadget patterns and searching strategies are the main reasons.
- We design a novel method to perform comprehensive and resilient searches for code-reuse attacks on obfuscated code. Our approach leverages symbolic execution and partial-order planning without any specific assumption on gadget patterns and searching strategies.

- We implement the proposed technique as a prototype tool and successfully use it to construct gadget chains on obfuscated programs, overcoming the limitations of other existing tools. The source code and evaluation benchmark are available at the following link Github.

II. Background

A. Software Obfuscation

In general, software obfuscation performs a semantics-preserving transformation on a normal program and outputs a complex form that is much more difficult to understand. It plays a crucial role in the protection of software intellectual property against malicious reverse engineering. Since Collberg's cornerstone obfuscation work [16], a variety of obfuscation strategies have been developed and packed into popular obfuscation tools such as Obfuscator-LLVM [1] and Tigress [2]. Several representative ones are listed as follows.

(1) Instructions Substitution replaces arithmetic or bitwise calculation with functionally equivalent but more complex instructions. For example, the xor calculation $\oplus$ can be substituted by other bitwise operations as follows.

$$a \oplus b = (\neg a \wedge b) \vee (a \wedge \neg b)$$

(2) Bogus Control Flow complicates the program's control flow by inserting conditional jumps that do not change the program semantics. For example, the condition is always-true, where the false branch is junk code, and the true branch is the original code. Usually, the path condition uses opaque predicates [17], [18], i.e., complex math expressions that always evaluated to the same value but are difficult to analyze.

(3) Control Flow Flattening breaks the normal program control flow into a dispatch structure inside a loop, where a control variable decides the program states [19].

(4) JIT Dynamic translates a function into a series of intermediate code representations. During runtime, this intermediate representation will be just-in-time compiled to machine code and start execution.

(5) Self-Modification inserts some special code at the head of the program. Each time the program runs, this code will change other parts of the executable code.

(6) Encode Data replaces integer variables, integer arithmetic, and integer and string literals with more complex expressions and opaque representations.

(7) Virtualization, a translation-based obfuscation, translates a program from one programming language to a heterogeneous language to hide the original program semantics. For example, The virtualization in Tigress translates a function written in C into a custom bytecode, which is further interpreted by a virtual machine during run-time. Consequently, the original program's behaviors are camouflaged within the complex virtual machine execution. Virtualization has been generally recognized as one of the most advanced obfuscation techniques to impede reverse engineering [20], [21].

B. Code-Reuse Attack

To date, code-reuse attacks still remain one of the most common software attacks [22]. The basic idea is to reuse small code snippets, i.e., "gadgets", in a benign program and combine them together to perform malicious behavior. Starting from re-using small-size gadgets ending with a `ret` instruction, Shacham's original work [23] has shown code-reuse attacks can be powerful, indeed Turing-complete. Since then, subsequent work has enriched the types of gadgets, such as using complex control flow structures [24], [25], call-preceded gadgets [22], multiple-architectures [26], [27], and dispatcher-gadgets [28], [29].

In the real world, a successful code-reuse attack aims to gain (root) control of the victim machine. They are often finished with manipulating the system-level resources such as:

- Invoke the system call `execve` to run shell `/bin/sh` or other programs on the victim machine controlled by the attackers.
- Invoke the system call `mprotect` to mark a page containing attacker-controlled content as executable and then redirect the program execution toward that tampered page.
- Invoke the system call `mmap` or `mremap` to map an attacker-controlled file as executable and then redirect the execution to that tampered file.

Building Code-Reuse Attacks. Traditionally, code-reuse attacks are manually constructed, which requires the attacker to be familiar with instruction architecture and have reverse engineering expertise. This manual procedure is tedious and error-prone, so people are seeking methods that automate the gadget-finding and payload-building procedure. These methods can be categorized into three types according to their design principles:

- *Pattern Matching.* ROPGadget [30] was one of the first code-reuse searching tools. It first searches for the occurrence of a set of pre-defined gadget patterns. Then the gadget candidates are chained together according to some built-in templates.
- *Symbolic Execution.* Angrop [31] leverages symbolic execution to recognize code-reuse gadgets inside binary code. It matches the symbolic execution result with the pre-defined semantics signatures of the gadgets. In this way, Angrop is more resilient than simple syntax-level pattern matching.
- *Program Synthesis.* SGC [32] is the state-of-the-art generic approach for building code-reuse payloads. It synthesizes logical formulas to encode the CPU and memory states at the beginning and end of a gadget chain. Then it queries an SMT solver to check whether a gadget chain satisfies the state transition.

III. Code-Reuse Attacks In Obfuscated Programs

In this section, we carefully investigate the code-reuse attacks in obfuscated programs and report interesting findings.

We are especially interested in answering the following questions.

- **Q1:** Does obfuscation introduce exploitable code-reuse gadgets?
- **Q2:** Can existing code-reuse exploit tools automatically construct attacks leveraging these new gadgets?
- **Q3:** What are the vital factors of constructing a feasible code-reuse attack in obfuscated programs?

We answer these questions by running various obfuscators on a benchmark and then using gadget-searching tools to scan them.

A. *Threat Model*

Our threat model involves an obfuscated binary file with a known stack memory write vulnerability, which means the author can write *any* value to the stack area. We also assume the attacker can find the gadget's address, e.g., through an information leak from certain vulnerabilities. Attackers are aware of these vulnerabilities that can be used as a starting point for code reuse attacks.

Note that we assume the memory vulnerabilities in the code/data presented in the binary file, probably in the obfuscation code, original program, or even the library functions. Identifying the memory vulnerabilities, which can be accomplished through a variety of automated vulnerability discovery tools [33]–[36] or through fuzzing [37]–[39], is not the purpose of this work.

Control Flow Integrity (CFI) [40]–[43], shadow stack [44], [45] are common techniques to counter code-reuse attack. Existing works already show that these defense methods are not safe [22], [25], [46]–[48], However, these techniques can hardly be used in obfuscated programs because the control flow in obfuscated programs is heavily mangled, which breaks the fundamental assumptions of these defense methods, leading to overwhelming false positives. Therefore, in this work, we assume that the obfuscated program is not protected a CFI or shadow stack system.

Address Space Layout Randomization (ASLR) seek to make code-reuse attack more difficult by adding randomness to the location of code while the program is running. However, numerous papers have shown that there are ways to use various side channels to leak ASLR information [49]–[52]. Therefore, we assume that the ASLR can be worked around or turned off.

B. *Experiment Setup*

Running this experiment requires a set of obfuscators, an obfuscation benchmark, and the code-reuse search tools.

Obfuscators. We use Tigress [2] and Obfuscator-LLVM [1], two popular obfuscators for software protection in academia and industry. They provide a variety of obfuscation schemes covering those presented in Sec. II.

More specifically, Obfuscator-LLVM adds obfuscation to LLVM intermediate representation. It provides three types of obfuscation strategies: instructions substitution, bogus control flow, and control flow flattening. Tigress is a source-to-source obfuscator for the C programming language. It directly translates the original C source code and produces the obfuscated C source code. In addition to the obfuscations in Obfuscator-LLVM, it supports more abundant obfuscation options, like virtualization, self-modification, and JIT dynamic.

To get the most obfuscated programs in our experiments, we turn on all possible obfuscation options provided by these tools. We also try each of them separately to observe their individual effect on code-reuse gadgets. Both Obfuscator-LLVM and Tigress provides command-line options to turn each obfuscation method on/off.

Benchmark. We adopt a third-party obfuscation benchmark from Banescu et al. [53]. This benchmark set: (1) includes sufficient diverse program functionalities and layouts; (2) provides scripts to automate the obfuscation and compilation procedure; (3) consists of C programs, which Tigress and Obfuscator-LLVM can correctly process.

Machine. Our experiments are performed on a server with the following configuration.

- CPU: Intel Xeon W-2123 4-Core 3.60GHz
- Memory: 64GB 2666MHz DDR4 RAM
- Hard Drive: 2.5TB SSD
- Operating system: Ubuntu 18.04

Code-Reuse Search Tools. We test three popular tools for searching code-reuse attacks: ROPGadget [30], Angrop [31], and SGC [32]. They are the representative tools corresponding to three categories shown in Sec. II-B.

The option `ropchain` in ROPGadget enables the ROP chain building. It also provides additional options like defining the number of bytes per gadget and we keep them at the default values. Angrop is implemented on top of the binary analysis platform Angr [54] with the VEX intermediate representation. The gadget chain building involves functions such as `find_gadgets`, `write_to_memory`, `set_regs`, and `print_payload_code`. After configuring these parameters, Angrop will automatically search the possible ROP chains. SGC builds logical formulas to encode the execution state and then queries an SMT solver to synthesize a valid gadget chain. We first modify the exploits configuration file based on our target binary to set up the preconditions and post-conditions, then run `extractor` to extract all functions and disassemble all gadgets. Then we use `synthesizer` with default settings to synthesize a gadget chain for those gadgets. We set up one hour for each benchmark program as the time-out threshold.

C. *Findings*

For every program in the benchmark, we first compile the normal version without any obfuscation. Next, we apply the obfuscators and output the obfuscated version. We use code-reuse search tools to find gadgets and try to build a valid gadget chain for a code-reuse attack.

Number of Gadgets. We observed that the number of gadgets increases substantially in the obfuscated programs. Fig. 1 shows a detailed comparison of the number of gadgets between

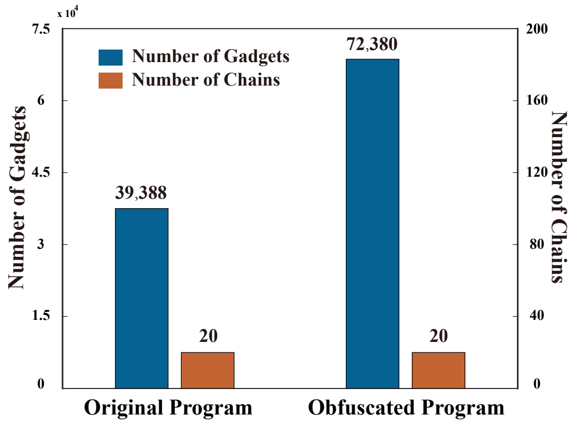

Fig. 1: Comparing the number of gadgets and the number of payloads (chains) found by existing tools from the original and obfuscated programs.

the original program on one type of obfuscated program (Instructions Substitution provided by Obfuscator-LLVM). The average increasing rate for all types of obfuscation methods in Sec. II-A is about 38.7%. All gadgets number are counted by ROPGadget because of its strong strength in searching gadgets. Therefore, obfuscation tools indeed introduce a large number of valuable gadgets into programs.

Gadget Chain. Surprisingly, existing tools cannot find more gadget chains in the obfuscated programs. Fig. 1 shows that these tools still generate the same number of gadget chains as the original programs. This result is counter-intuitive because the obfuscated programs include so many more gadgets that the tools should be able to generate more chains. The conflict leads us to inspect how those automated tools construct gadget chains and discover their limitations.

D. *Limitations of Existing Chain-Building*

A successful code-reuse attack triggers one of the system calls described in Sec. II-B. To achieve this goal, the search tool needs to find proper gadgets *to set up the parameters for the system call*. For example, suppose the attacker wants to call `execve` to spawn a shell. Under the x86 calling convention, the system call number `0x3b` must be placed in the `rax` register. Registers `rdi`, `rsi` and `rdx` store the arguments of `execve`. `rdi` stores the starting address of a character string. Usually, this is the full path of a program that the attacker would like to run, e.g., `"/bin/bash"`. The character string can be part of the payload written into the stack; `rsi` saves the argument list passed to the executable program; `rdx` saves a pointer pointing to the environment variables, which is usually set as a NULL pointer.

We identify the following limitations of the chain-building procedure of existing automatic searching tools.

Strict Syntax Pattern Matching. Some tools rely on strict, hard-coded patterns on the syntax level to find gadgets. A chain is formed only when all the exact gadgets are found in the targeted binary. ROPGadget is in this category. As an example, Fig. 2(a) is one strict syntax pattern. The chain built from this pattern triggers an `execve` through the interruption `int 0x80`. `reg2` stores the system call number. It is initialized as 0 by the `xor reg2, reg2` gadgets and increases to the required value by the `inc reg2` gadget. A valid chain can be successfully built only when all these gadget patterns are matched in the program, so it largely limits the possibilities of constructing valid chains.

Semantic Pattern Matching. To overcome the weakness above, some other searching tools adopt semantic patterns, which allow users to describe the patterns or the final attacking goal. Then the tool will automatically search for chains to satisfy the description. Both Angrop and SGC leverage semantic patterns.

Although appearing to be more flexible than the syntax level pattern matching, these pre-defined descriptions and the matching process still heavily impede the chain-building process. For example, suppose the goal is to assign the value `0x3b` the register `rax`. Semantic pattern matching only checks common gadget sequences like "`pop rax; ret;`". However, the matching fails if the target binary program does not include these types of gadgets. Fig. 2b shows another way to build the value `0x3b` by adding `0x30` and `0xb`. However, semantic-pattern matching is incapable of building gadget chains like this.

Searching Strategy. We also find that current tools only aim at building simple and short chains [30], [31], avoiding searching for more complex possibilities. They limit the size of the gadgets and the number of gadgets in each chain. This greedy strategy may be good for performance, but it misses many opportunities for building feasible chains. For example, Angrop only uses the pure `pop` and `ret` gadgets for assignments, ignoring other alternative gadgets that contain some irrelevant instructions.

Gadgets from Obfuscation. Another important observation is that existing tools rarely use the new gadgets injected by obfuscations. We found that this is because obfuscations introduce many complex gadgets that cannot be processed by existing tools. For example, many gadgets in obfuscated programs contain conditional jumps and indirect jumps. Table I shows that the number of gadgets with conditional jumps after obfuscation is 60% more than before, and the number of gadgets with indirect jumps after obfuscation increased 50% more than before. We carefully studied the common gadgets used by the ROPGadget, Angrop, and SGC in gadget chains, and we found that none of them used any gadgets with conditional jumps or indirect jumps. ROPGadget and Angrop only use gadgets ending with "ret". Some gadgets from SGC's results include indirect jumps, but no direct jumps or conditional jumps are found.

E. *Conclusion*

In summary, to answer the three questions at the beginning of this section:

TABLE I: The types of the gadgets and average total number of each type found by ROPGadget in original programs and obfuscated programs. **U** means unconditional; **C** means conditional; **DJ** means direct jump; **IJ** means indirect jump. The **Original** means the program before obfuscation. **Obfuscated** means the program after obfuscation. **IR** means increasing rate.

Gadget Type	Description	Example	Original	Obfuscated	IR
Return	The gadget end with a `ret` instruction	pop rax; ret;	9,231	16,868	82.70%
UDJ	Jump to a constant address without condition	pop rax; jmp 0x401235;	40,034	57,110	42.65%
UIJ	Jump to a register or memory without condition	pop rbp; mov edi, 0x601030; jmp rax;	3,216	5,255	63.40%
CDJ	Jump to a constant address with condition	test eax,eax; jg 0x14b3; jmp 0x13fd;	5,294	8,727	64.84%
CIJ	Jump to a register or memory with condition	je 0x4003e2; call rax;	1,988	3,184	60.16%

```
// Set syscall number
xor eax, eax
ret
inc eax (add eax, 1)
ret

// Set syscall arguments
pop ebx
ret
pop ecx
ret
pop edx
ret

// Trigger syscall
int 0x80
```

(a) Syntax pattern matching.

```
// The logic pattern 1
rax = 0x3b
rdi = 0x4018b2
rsi = 0x0
rdi = 0x0

// The logic pattern 2
rax = 0x30
rbx = 0x0b
rax = rax + rbx (rax = 0x3b)
rdi = 0x4018b2
rsi = 0x0
rdi = rsi (rdi = 0x0)
```

(b) Semantic pattern matching.

Fig. 2: The pattern matching methods for building gadgets. ROPGadget uses syntax patterns and Angrop uses semantic patterns.

- Software obfuscation methods introduce a large number of code-reuse gadgets into a program. Existing gadget-detection tools can easily find these gadgets indicating high risks of potential code-reuse attacks.
- However, existing searching tools have very limited capability to chain these gadgets into valid payloads on certain types of obfuscation.
- The reasons are due to the strict syntax and semantic patterns, searching strategy, and the diverse gadget types in the obfuscated programs.

IV. Gadget-Planner

To overcome the limitation of existing code-reuse exploitation tools, we propose a new method, Gadget-Planner, to comprehensively explore code-reuse attacks in an executable file. Our method does not rely on any syntax/semantic patterns. This section first discusses the overall workflow and then elaborates on each component.

A. Overview

The core idea of Gadget-Planner is to leverage the power of symbolic execution and partial-order planning, an artificial intelligence (AI) method to find a sequence of actions from one system state to the desired goal state [55], [56]. Given a program binary, we first perform symbolic execution and constraint solving to generate the "semantic metadata" for every gadget candidate. Next, a planner selects and assembles those gadgets, which attempts to construct a complete gadget chain forming a successful code-reuse attack.

Strength. Our method overcomes the limitation of previous works from two aspects: (1) Gadget-Planner does not rely on any syntax/semantic patterns. The planner can construct the gadget chain freely based on any gadgets collected from the binary program. (2) Gadget-Planner can handle all types of gadgets in Table I, regardless of direct/indirect and conditional/unconditional.

Challenge. The major challenge rises from the enormous number of possible gadgets in a binary program. As Fig. 1 shows, obfuscated programs include more possible gadgets than original programs. Blindly searching among those huge amounts of gadgets will inevitably cause the state-explosion problem.

Gadget-Planner has two strategies to alleviate this challenge. First, we only apply symbolic execution and constraint solving on short code snippets such as a part of basic block, avoiding the heavy overhead from running them on large programs. Second, several heuristic search optimizations are adopted to help guide the planner in gradually assembling smaller gadgets toward the final attacking goal.

Workflow. As shown in Fig. 3, Gadget-Planner's overall workflow contains four main stages.

1) *Gadget Extraction.* Gadget-Planner extracts gadgets from the target binary code. It seeks to decode from the valid starting position of each basic block in the control flow graph of binary code until reaching a jump instruction. From the list of potential gadgets, Gadget-Planner determine whether the gadgets end in a controllable indirect jump or a direct jump. All the gadgets ending with a direct jump are joined with the gadget starting from that location; the merged gadget will be treated as a single gadget.
2) *Subsumption Testing.* This stage winnows the list of gadgets down to a minimal subset. Gadget-Planner run symbolic execution on the gadgets and use solvers to check their equivalence. Only one gadget is kept in its equivalent class.
3) *Partial-order Planning.* Gadget-Planner uses partial-order planning to chain the identified gadgets together. Starting from the final attacking state, the planner attempts to build a gadget chain, essentially in a backward direction, until it reaches a valid initial memory state which can be injected into stack memory.
4) *Post-processing.* Last, Gadget-Planner transforms the planning result into a memory payload, which can be directly

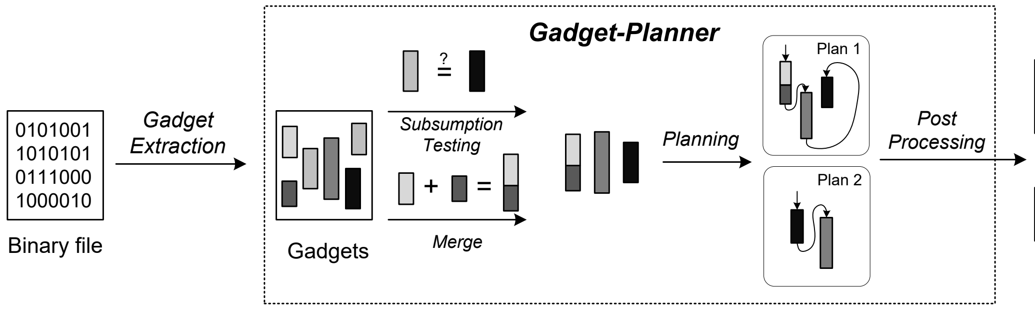

Fig. 3: High level overview of Gadget-Planner's architecture.

TABLE II: The record for describing a gadget.

Field	Description
`len`	Measures the gadget length in bytes
`location`	The address of the first instruction in each gadget in the binary file
`jmp-type`	The jump instruction at the end of the gadget
`clob-reg`	The registers whose contents are overwritten
`ctrl-reg`	The registers can be controlled through the gadget
`pre-cond`	Symbolic constraints required for gadget to run successfully, e.g., register contents must be set to specific values.
`post-cond`	Symbolic constraints representing the effects of the gadget

placed on the victim's stack to trigger the desired code-reuse exploitation.

B. Gadget Extraction

Taking a binary file as the input, our first task is to seek available gadgets in the file as the candidate for constructing a code-reuse attack. We disassemble the raw binary file and start searching at any position within a basic block. This strategy can detect unaligned instructions and use them for future chain-building procedures. It also accepts parameters to ignore the first N instructions and search from an arbitrary position in the middle of a basic block.

The gadgets are further analyzed by symbolic execution to create the pre-condition and post-condition. Generally, a pre-condition describes the condition that guarantees the gadget's functionality. A post-condition is related to the execution result of this gadget.

For each gadget, we create a record storing information like gadget length, location, jump type, the registers being changed or controlled, pre-condition, and post-condition. A detailed description is shown in Table II. These records contain sufficient information, so all future analyses are performed on them.

Conditional Jump. Processing conditional jumps are a distinct feature of our gadget extraction stage. Gadget-Planner can extract the gadgets with both indirect and direct conditional jump, e.g., `jg <address>` or `jne ebx`. In contrast with our method, existing gadget-searching tools only consider gadgets ending with unconditional jumps. This feature significantly enriches the gadget pool, which will be further processed for exploitation generation.

```
xor rax, rax
ret
```

(a)

```
inc rax,
sub rdx, rbx
jnz 0x1234
ret
```

(b)

```
mov  rax, rbx
test rcx, rcx
jz   0x5678
...

0x5678:
pop  rbx
ret
```

(c)

Fig. 4: Conditional jumps in gadgets.

TABLE III: The record for the conditional gadgets in Figure 4(b).

Field	Value
`len`	16
`loc`	0x4321
`jmp-type`	`ret`
`clob-reg`	`rax, rdx`
`ctrl-reg`	`rax`
`pre-cond`	`rbx == rdx`
`post-cond`	$\texttt{rax}_{after} == \texttt{rax}_{before} + 1$

The jump condition is placed into the precondition extracted by symbolic execution. When constructing the gadgets chain, these gadgets are used if the precondition is satisfied.

Figure 4 compares the normal gadgets with two different situations of gadgets with conditional jumps. Figure 4(a) first presents traditional return-ended gadgets without any conditional jumps. Figure 4(b) shows one gadget with a conditional jump `jnz 0x1234` in the middle. To ensure the functionality of this gadget, the conditional jump must *not* be taken, which means, the condition "`rdx-rbx != 0`" must be false. Therefore, we must guarantee "`rdx == rbx`" to use this gadget. Figure 4(c) shows another situation also involving conditional jumps, where the first half of the gadget ends with a conditional jump that must be taken to jump to the second half. This time, the jump condition must be true to trigger the jump, i.e., "`rcx == 0`" must hold.

Table III shows the value of each field in the record of Figure 4(b)'s gadget. These values provide an abstract description of the gadget's functionality. The pre-condition and post-condition usually involve two sets of symbolic variables representing the program states which contain register and memory symbol values before and after the gadget execution. $\texttt{rax}_{\text{before}}$ is the value in the `rax` register before the gadget execution. For simplicity, Table III uses a C-like syntax to present the constraints.

Those constraints generated by memory operations like `mov` between registers and memory require a value working as a "pointer" to a readable or writable memory area. To express this, we add a new type in the constraint description language called `POINTER`, to distinguish it from normal bit-vectors. For example, when triggering the execve() syscall, we need to pass an argument list to this call; those are all passed with a pointer type, so under our new type of constraint, the target state of calling an execve() can be described as `{'rax': 57, 'rdi': path/to/executable, 'rsi': Pointer(to=0), 'rdx': Pointer(to=0)}`.

Since analyzing whether an arbitrary address is readable or writable is difficult in general, we restrict these reads or writes to occur only on the stack if the value of a register depends on a read from memory that we control (e.g. the payload area on the stack), the after the variable is left unconstrained so that it is free to take on whatever value is necessary for the rest of the plan.

Unconditional Direct Jump. Our method can process those gadgets ending with direct jumps as well, like `jmp 0x1234`, and use them to construct chains. The keystone is to check the instructions at the target address. We follow the direct jump and merge the gadget ending with a direct jump with the one at the targeting address. The two gadgets are considered together in the following stages.

C. Subsumption Testing

The gadget extraction stage usually produces an enormous number of available gadgets from an obfuscated program. We conduct subsumption testing to avoid repeat searching among the gadgets with the same functionality. The high-level idea is to remove redundant equivalent gadgets because if two gadgets perform the same function, then we only need one copy of them. For every pair of gadgets A and B, if A can subsume the semantics of B, then B is deemed a redundant gadget that can be safely removed without undermining the expressiveness of the whole gadget collection from this binary.

Let gadget g_1's pre-condition as pre_1 and post-condition as $post_1$. Similarly, let gadget g_2's pre-condition as pre_2 and post-condition as $post_2$. To decide whether g_1 exhibits semantics subsuming g_2, we check if pre_1 is a superset of pre_2 and if $post_1$ equals $post_2$. This means g_1 is functionally equivalent with g_2 and on a looser pre-condition. For example, if a gadget has a pre-condition `rbx >= rdx`, which is a superset of `rbx == rdx` as in Table III, and the post-conditions are equivalent, then it subsumes the gadget in Fig. 4(b).

To this end, we construct the constraint (1) as below to perform the subsumption testing. The first clause asserts that pre_2 is a subset of pre_1, while the second clause asserts that $post_2$ equals $post_1$. If it is true, then g_2 can be safely removed.

$$(pre_2 \rightarrow pre_1) \wedge (post_1 = post_2) \tag{1}$$

The subsumption testing guarantees to keep exactly one gadget for one functionality. If there is a tie (i.e. g_1 subsumes g_2 and g_2 subsumes g_1), it means g_1 and g_2 are equivalent. We randomly keep one of them in the gadget pool.

D. Partial-order Planning

With the gadgets processed from previous phases, we now apply a technique called *partial-order planning* to find a sequence of gadgets from the library that forms a valid code-reuse attack. The goal is to trigger one of the attacks in Sec. II-B and we are seeking a plan to achieve this goal, utilizing the gadgets as actions in the plan.

Background. Planning is a traditional area of artificial intelligence focusing on finding a sequence of actions that takes a system from a specified initial state to a desired goal state [55], [56]. Planning methods have been widely adopted in robotics [57] and other artificial intelligence applications [55].

One common planning problem is finding a path in a directed graph of discrete states [58]. More formally, an arc from state s_1 to state s_2 exists if there is a valid action in state s_1 that results in state s_2. Hence a state-space search problem can be described as a tuple $\langle S, s_0, S_G, A, c(a,s)\rangle$ where:

- S is a finite set of states s,
- $s_0 \in S$ is the starting state,
- $S_G \subseteq S$ is a set of goal states,
- $A(s) \subseteq A$ is the set of actions applicable to each state $s \in S$, and
- $c(a,s)$ calculates the cost of performing action a at the state s.

The planning procedure searches the graph for a *solution*, i.e., a state's transition trajectory from the starting state to a goal state. More accurately, a solution is a sequence of state transitions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots, \xrightarrow{a_{n-1}} s_n$ so that $s_n \in S_G$ and $a_i \in A(s_i)$. The solution is *optimal* if the total cost $\sum_{i=0}^{n-1} c(a_i, s_i)$ is minimum.

An intuitive search procedure is a breadth-first search, which first examines every applicable action at the initial state s_0, then every applicable action at the successor states, and so on until a goal is reached. For a specific problem, we often have heuristics that give an approximation of how far away from a goal a state is. They can guide the graph search to find a goal more quickly.

Gadget-Planner. In our planning problem, each discrete state corresponds to a specific combination of values in registers and memory. Triggering a system call like `execve` is a goal state. Each gadget corresponds to an action that transitions the state of the system. The planner's task is to find a sequence of gadgets that takes the system from its initial state to a goal state. In partial-order planning, a plan is not required to be a totally ordered sequence; each gadget must come before those that depend on it, but non-interfering gadgets may be allowed to occur in either order relative to one another. Because a single partial order is compatible with many total orderings, the space of partially-ordered plans is smaller than that of sequences, allowing a faster search for a valid plan. Once found, a partially-ordered solution plan is linearized to form the attack.

Formally, the planning process operates on a 5-tuple problem state $\langle \alpha, \beta, \gamma, \delta, \epsilon \rangle$, each of which represents a (possibly incomplete) attack plan:

- α is a set of gadgets $\{g_1, g_2, \ldots, g_n\}$ selected so far for the exploit.
- β is a set of gadget orderings. Each ordering $\langle g_i, g_j \rangle$ describes a data dependency requiring that the gadget g_i must precede g_j. One common example is that g_j overwrites the data written by g_i.
- γ is a set of gadget pairs describing the causal links. A *causal link* between two gadgets g_i and g_j means that the post-condition of g_i fulfills the pre-condition of g_j.
- δ is a set of pre-conditions that have not yet been fulfilled.
- ϵ is a set of causal links that are currently *unsafe*, meaning that there exists a gadget $g_k \in \alpha$ that would negate the previous post-condition of the causal link and could be ordered between g_i and g_j with violating some gadget ordering.

Essentially, the planning process is a backward search from the goal, i.e., one code-reuse attack described as a constraint. The planner iteratively selects a plan, which is a set of gadgets that can result in the searching goal, from its current set of partial incomplete plans, selects an open pre-condition from that plan's δ, and tries to find gadgets from α that can fulfill it. If it fails, the plan is a dead-end and can be discarded. Otherwise, we generate a successor plan for each gadget whose post-condition fulfills the open pre-condition. Then the fulfilled pre-condition is removed from δ, and the new gadget's pre-condition is added. The new causal link is added to γ. The data dependency between the new and existing gadgets is analyzed, and β is updated accordingly.

Note that the gadget pairs in β and γ represent a *partial-order*, so the new gadgets can be inserted between these pairs to construct the gadget chain. This design gives the planner abundant freedom to build diverse gadget chains.

Plan Completion. When there are no unfulfilled pre-conditions, i.e. $\delta = \emptyset$, the new successor plan represents a complete plan that leads to a valid code-reuse exploit. Otherwise, the new partial plan is added to the planner's set of incomplete plans based on the order of the priority queue, which is sorted by some heuristic values, such as the number of remaining dependencies. Usually, incomplete plans with the fewest remaining dependencies are preferred. Gadget-Planner does not stop when finding one gadget chain; it keeps searching for more diverse gadget chains as many as possible.

Unsafe Causal Link Elimination. Generating a successor plan in this way may result in a plan where $\epsilon \neq \emptyset$. We then need to eliminate all the unsafe causal links in ϵ by finding a partial order that doesn't violate any of the dependencies of the gadgets in β. To test the dependencies, we check every register's dependency on the current gadget and compare their values with all previous gadget's register values. The order must be changed if the current value changes any previous

Algorithm 1 Gadget-Planner algorithm.

```
1: Input: A goal $G$, and a library of gadgets $L$.
2: function SEARCH($G, L$)
3:   $Q \leftarrow$ InitPriorityQueue()
4:   $P \leftarrow$ InitPlan($G$)
5:   $Q$.Add($P$)
6:   while $Q$ is not empty do
7:     best $\leftarrow$ pop($Q$)
8:     precond $\leftarrow$ GetPreCond(best)
9:     gadgets $\leftarrow$ PickIfSatisfy(precond, $L$)
10:    for $g \in$ gadgets do
11:      plan $\leftarrow$ AddToPlan(gadget, best)
12:      if $\exists$ constraint UNSAT in plan then
13:        continue
14:      else if Dependency(plan) = $\varnothing$ then
15:        Output plan
16:      end if
17:      Add plan to $Q$
18:    end for
19:  end while
20:  return Failure
21: end function
```

value. If a dependency cannot be solved by reordering the gadgets, the plan is invalid so it will be discarded from the current planning queue.

Planning Algorithm. Putting these ingredients together, Algorithm 1 presents the pseudo-code for the whole planning process. The for loop beginning on line 8 enumerates the possible successor plans. Since the search starts from the desired exploit and works backward, we first consider the plan consisting only of the pre-condition to trigger the exploit. The algorithm starts by inserting that initial state into a priority queue (lines 4 and 5). We use a greedy best-first search, meaning that the priority queue is ordered by the heuristics specified below. Each iteration of the search loop at line 6 pops a partial plan from the priority queue, generates the successor plans, and inserts them into the queue. Lines 9 through 15 explain how the successor plans are evaluated and added to the queue. The whole procedure continues and prints out a complete plan (line 13) until the search space is exhausted and the queues emptied (line 18), so Gadget-Planner can find multiple different plans from the pool of candidate gadgets.

Heuristics. The priority queue is sorted based on the following heuristics. The weights are from high to low.

- *Number of remaining pre-conditions.* The search prefers a partial plan with fewer remaining pre-conditions. This leads the planner to first check the plans that appear almost complete.
- *Number of constraints.* We prefer plans with fewer constraints because they are likely to be easier to satisfy and hence cheaper to enumerate and elaborate. Similarly, the planner prefers fewer symbolic variables.

- *Complexity.* The planner prefers simpler gadgets because it is usually easier to build a chain using simple increments and mathematical operations combinations.

E. Post-processing

After finding the valid plans, we get a partial-order sequence of gadgets. As the last step, we generate a stack payload for this gadget sequence. The payload is a binary sequence that can be placed on the stack when the executable file is running to trigger the attack. The partial order gives flexibility in changing the order of some gadgets in the payload.

V. Implementation

The implementation of Gadget-Planner contains approximately 1,500 lines of Python code on the basis of the Angr binary code analysis framework [54]. Angr facilitates lifting binary code from various architectures into the VEX intermediate representation and performs symbolic execution. Gadget-Planner employs the popular constraint solver Z3 [59] to define symbolic variables in bit-vectors and solve constraints.

Gadget-Planner can search and generate payload for triggering the three types of attacks in Sec. II-B. These gadget chains precisely set the contents of specific registers and then jump to a `syscall` instruction.

Inside Gadget-Planner, gadgets are represented as a separate data structure to facilitate passing gadget information and referencing the same gadget multiple times unambiguously during planning.

In particular, Gadget-Planner represents the gadget library as a dictionary keyed on the register name, i.e., indexing the available gadgets by the registers they affect. This implementation allows the planning phase to easily select the gadgets that are relevant for achieving a given goal. Selecting gadgets in this way, instead of considering all gadgets in all states, substantially reduces the branching factor of the search.

VI. Evaluation

This section aims to answer four research questions (RQs) regarding Gadget-Planner:

- **RQ1:** Comparing to existing work, how effective is Gadget-Planner in constructing code-reuse gadgets from obfuscated and un-obfuscated programs? *(effectiveness)*
- **RQ2:** How does the code obfuscation affect code-reuse attack chains in obfuscated programs? *(effectiveness)*
- **RQ3:** Can Gadget-Planner generate code-reuse payloads for obfuscated real-world programs? *(practicability)*
- **RQ4:** How much overhead does Gadget-Planner introduce? *(performance)*

As the answer to RQ1, we apply Gadget-Planner to the same obfuscation benchmark in Sec. III, then report the result of constructing valid gadget chains on obfuscated and un-obfuscated programs. For RQ2, we run Gadget-Planner on obfuscated programs with different obfuscation methods to check the generated gadget chains. To answer RQ3, we successfully build a code-reuse payload for a real-world obfuscated application. In response to RQ4, we report Gadget-Planner's searching time and memory usage.

A. Code-Reuse Attacks in the Benchmark

We apply Gadget-Planner to the benchmark programs (original and obfuscated) and compare the result with the peer tools. The time threshold is set to 2 hours for each program.

Number of Gadgets. The "Number of Gadgets" columns in Table IV show how many gadgets are collected by each tool. We count the total number in the gadgets pool and the number of gadgets in the chain. The data shows that the peer tools collect tens of thousands of gadgets but can only use very few of them when building the chain. On the contrary, Gadget-Planner's gadgets collection is less than the peer tools because of the gadget combination and subsumption testing. Concatenating smaller gadgets to larger gadgets enriches the functionalities of the gadgets inside the searching pool, and the subsumption testing filters out the gadgets with the same behavior. Also, the planning algorithm does not assume any particular patterns, so it can comprehensively build different gadget chains. Consequently, Gadget-Planner efficiently constructs a smaller gadget pool and builds more chains from them.

Number of Code-Reuse Payload. Gadget-Planner finds many more code-reuse payloads than the peer tools. Table IV shows the experiment result. On the obfuscated programs, Gadget-Planner can find 30X more payloads than ROPGadget, 10X than Angrop, and 2X than SGC.

We observe that the factors limiting existing tools are strict patterns, searching strategies, and complex obfuscated gadgets. More specifically, ROPGadget only matches the `execve` pattern, which is hard-coded as syntax-level searching and thus strictly limits the capability for building the payloads. Similarly, all gadget chains constructed by Angrop share identical patterns. For instance, it only uses `pop reg; ret;` to assign a value to registers regardless of all other equivalent gadget variants. SGC uses a gadget selection function to reduce the search area, so the gadget candidates pool is similar in different searches. Distinct from all these existing tools, Gadget-Planner resiliently searches gadgets and builds code-reuse payloads broadly for obfuscated programs.

Gadget Diversity and Complexity. We further compare the payloads generated by Gadget-Planner with others and find that Gadget-Planner can build the most diverse code-reuse chains. Table V compares the percentage of the four types of gadgets that appeared in the chains: return, indirect jump, direct jump, and conditional jump. ROPGadget and Angrop only use gadgets ended with return instructions. SGC can construct chains with gadgets ended with return and indirect jumps. However, none of them handles direct jumps or conditional jump instructions. Instead, Gadget-Planner uses all types of gadgets to construct a diverse collection of chains.

Unlike existing exploitation tools that only use simple gadgets, Gadget-Planner can build diverse chains with complex gadgets. More precisely, the chains include various gadgets, and each gadget contains more instructions. To quantify the diversity and complexity, we measure two metrics: the gadgets' length and the chains' length. Overall, longer gadgets

TABLE IV: A comparison of the number of gadgets and payloads between Gadget-Planner and the peer tools on obfuscated and non-obfuscated programs. The parenthesized numbers count the number of payloads newly introduced by the obfuscation.

Obfuscation	Number of Gadgets (Total/Used)				Attack	Number of Payloads			
	ROPGadget	Angrop	SGC	Gadget-Planner		ROPGadget	Angrop	SGC	Gadget-Planner
Original	39,388 / 6	12,958 /18	2,460 / 61	**885 / 40**	`execve`	1	1	6	**8**
					`mprotect`	-	1	5	**6**
					`mmap`	-	1	5	**6**
					Total	1	3	16	**20**
LLVM-Obf	62,989 / 6	33,484 / 18	4,779 / 61	**1,490 / 185**	`execve`	1 (0)	1 (0)	6 (0)	**15 (7)**
					`mprotect`	-	1 (0)	5 (0)	**10 (4)**
					`mmap`	-	1 (0)	5 (0)	**12 (6)**
					Total	1 (0)	3 (0)	16 (0)	**37 (17)**
Tigress	50,459 / 6	27,036 / 18	3,911 / 61	**1,224 / 205**	`execve`	1 (0)	1 (0)	6 (0)	**16 (8)**
					`mprotect`	-	1 (0)	5 (0)	**16 (10)**
					`mmap`	-	1 (0)	5 (0)	**9 (3)**
					Total	1 (0)	3 (0)	16 (0)	**41 (21)**

TABLE V: The gadget chain properties. The gadget length and chain length are measured by the average number of instructions. The remaining columns describe the percentage of each gadget type in the chain. The "Ret", "IJ", "DJ", "CJ" refer to return type, indirect jump, direct jump, and conditional jump.

Tool	Gadget Len	Chain Len	Ret	IJ	DJ	CJ
ROPGadget	2.1	12.6	100%	0%	0%	0%
Angrop	2.3	13.8	100%	0%	0%	0%
SGC	5.8	23.2	68%	32%	0%	0%
Gadget-Planner	6.7	33.5	38%	10%	12%	40%

and chains indicate that the tool can handle more complex gadgets and search deeply in the gadget space. Table V shows that Gadget-Planner builds longer gadget chains using larger gadgets than the peer tools do. The average chain length is 1.5 times than SGC and 3 times than ROPGadget and Angrop.

Unobfuscated Programs. The "Original" row in Table IV compares Gadget-Planner and other tools on the unobfuscated benchmark programs. The result shows that Gadget-Planner can also build more gadget chains on unobfuscated programs. The reason is that Gadget-Planner does not rely on any specific structures inside obfuscated programs. Instead, the symbolic execution and automated planning in Gadget-Planner capture the generic features in code-reuse gadgets and chains.

B. Obfuscation and Code-Reuse Risks

The parenthesized numbers at the rightmost column in Table IV shows the number of chains that are newly introduced by the obfuscations. Gadget-Planner is the only tool can find these gadget chains.

Another important finding is that the number of code-reuse attacks varies on the obfuscation methods, which means some obfuscations bring more code-reuse attack risks than others. Fig. 5 shows the number of payloads on different obfuscations, where the top three, bogus control flow, control flow flatten, and virtualization, are marked as red. They introduce a high code-reuse risk because they inject a huge number of jump instructions that are easier to use for the attack. Therefore, users must cautiously adopt these three obfuscations.

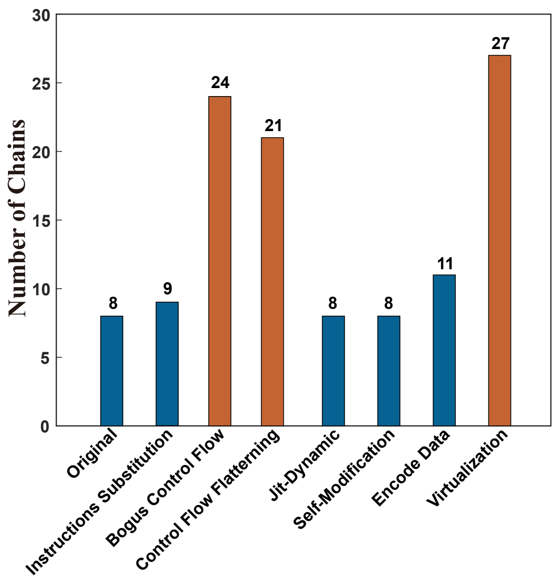

Fig. 5: Comparing the number of chains from the original program and different types of obfuscated programs.

C. Real-World Applicability

To evaluate Gadget-Planner in real-world scenarios, we run Gadget-Planner with the peer tools on SPEC 2006 benchmark programs [60]. Also, we trigger a code-reuse attack on a buggy version of *netperf* [61] using the payload constructed by Gadget-Planner.

SPEC Benchmark. We apply the obfuscators to all programs in the SPEC benchmark and successfully produce four obfuscated programs from LLVM-Obfuscator and two from Tigress. The main reasons for the failed cases are: 1) some programs are written in Fortran and thus cannot be processed by LLVM-Obfuscator or Tigress; 2) several old programs depend on obsolete libraries and we encounter difficulties when building them on modern machines; 3) some data structures, like the builtin type `_Complex`, are not supported by LLVM or Tigress. We apply Gadget-Planner and other peer tools to the non-

TABLE VI: Comparing the number of gadgets and chains between Gadget-Planner and the peer tools on the original and obfuscated programs from the SPEC benchmark. “RG” means ROPGadget and “GP” is Gadget-Planner. “-” means Tigress fails to obfuscate the program.

Benchmark	Original					LLVM-Obf					Tigress				
	Gadgets	RG	Angrop	SGC	GP	Gadgets	RG	Angrop	SGC	GP	Gadgets	RG	Angrop	SGC	GP
401.bzip2	3,896	0	0	2	3	13,825	0	0	2	5	3,667	0	0	2	3
429.mcf	2,501	0	0	0	3	14,891	0	0	1	8	3,158	0	0	1	1
445.gobmk	32,847	0	1	4	7	33,319	0	1	4	12	-	-	-	-	-
456.hmmer	10,964	0	1	4	5	41,840	0	1	4	7	-	-	-	-	-

Gadget 1:
```
0x4a8462:   pop     rsi
            test    rax, rax
            jg      0x483848
            add     rbx, 4
            pop     rbx
            pop     rbp
            ret
```
Gadget 2:
```
0x4707f2:   pop     rax
            test    rax, rax
            je      0x42cebd
            pop     rdi
            add     rsp, 8
            jmp     rdi
```
Gadget 3:
```
0x484af5:   pop     rdx
            cmp     rbx, 0x190
            jne     0x484960
            add     rsp, 4
            pop     rbx
            ret
```
Gadget 4:
```
0x4ab46c:   call    rbx
```

Fig. 6: A gadget chain from `429.mcf` built by Gadget-Planner. It cannot be built by any existing tool.

obfuscated and obfuscated programs. The search time is set to 2 hours for each program.

The result in Table VI shows that Gadget-Planner outperforms the peer tools in terms of finding more attack payloads from both the original and the obfuscated programs. Existing tools cannot find the exploit chain because their search strategies are too simple, and the alternative gadget pool is not diverse enough. ROPGadget and Angrop heavily depends on strict patterns. Once a gadget in the pattern is missing, the whole search will fail. For example, if `pop rdx; ret` does not exist in the current program, Angrop fails to set a value to this register. Due to using some gadgets with an indirect jump, SGC will find exploit chains that the other two existing tools cannot find in some situations.

Gadget-Planner’s diverse gadget pool and flexible search strategy enable it to find many more chains that other tools cannot find. Fig. 6 shows a chain example built by Gadget-Planner from the `429.mcf` in SPEC. Note that none of the other tools find any chains from that program. After carefully checking the chain-building procedure, we find that the reason is the lack of one special gadget `pop rsi; ret`, so the target value cannot be correctly assigned to `rsi`. On the other hand, Gadget-Planner successfully connects two separate gadgets

```
void break_args(char *s, char *arg1, char *arg2)
{
    char *ns;
    ns = strchr(s,',');
    if (ns) {
        *ns++ = '\0';
        while ((*arg2++ = *ns++) != '\0');
    }
    else {
        ns = s;
        while ((*arg2++ = *ns++) != '\0');
    };
    while ((*arg1++ = *s++) != '\0');
}
```

Fig. 7: The stack overflow vulnerability in the `break_args` function from netperf.

together with a conditional jump, such as the `Gadget 1` in Fig. 6, achieving the same functionality as the missing specific pattern.

Netperf Case Study. To demonstrate the payloads constructed by Gadget-Planner is useful in practice, we conduct a case study on *netperf*, a software providing network bandwidth testing between client and server. The version 2.6.0 contains a stack overflow vulnerability when running the client with command line arguments `'-a'`. The function `break_args` copies the contents from `optarg` to `arg1` and `arg2` without length checking, as shown in Fig. 7. This vulnerability allows attackers to inject any length of the payload of gadget chains to get a system shell and, thus is suitable for triggering code-reuse attacks. We assume ASLR and canary protection have been disabled.

Gadget-Planner successfully finds 16 gadget chains from the obfuscated program. In our experiment, we use LLVM-Obfuscator with all three obfuscation strategies to generate the obfuscated program. Fig. 8 shows one example chain. It spawns a shell after being passed as an argument when running `netperf` with the `'-a'` option. It triggers the syscall `execve(&"/bin/sh", 0, 0)` to open a shell. The registers `rdi, rsi, rdx` hold the parameters being passed to `execve` and `rax` holds the system call number `0x3b`. These are the goal for our planning procedure. After the goal state is set up, our planner focuses on those gadgets which are changing specific registers’ values in the current state, then adds the new pre-conditions in the successor plans for the next round of planning until all needed register states fulfill the goal state.

```
Gadget 1:
0x49cc0a:   pop     rax
            test    rax, rax
            je      0x49cc7d
            add     rsp, 8
            jmp     rax
Gadget 2:
0x61482f:   pop     rsi
            pop     rdi
            test    rax, rax
            jg      0x61480a
            add     rsp, 8
            pop     rbx
            pop     rbp
            pop     r12
            pop     r13
            ret
Gadget 3:
0x634745:   pop     rdx
            test    r8d, r8d
            je      0x634750
            add     rsp, 8
            ret
Gadget 4:
0x4b37d8:   pop     rax
            ret
Gadget 5:
0x4b3a33:   call    rbx
```

Fig. 8: A gadget chain built by Gadget-Planner from *netperf*.

TABLE VII: The performance of every component in Gadget-Planner and the peer tools when analyzing the obfuscated *netperf*.

Tools	Stages	Time (S)	Memory (GB)
Angrop	`Gadgets Finding`	482	4.45
	`Chain Generating`	9	2.11
	Total	491	-
SGC	`Disassembly`	2,445	5.33
	`Chaining`	3,600	11.36
	Total	6,045	-
GP	`Gadget Extraction`	2,223	4.16
	`Subsumption Testing`	2,104	5.73
	`Planning`	1,771	8.43
	Total	6,098	-

D. *Performance*

We run Gadget-Planner and the peer tools on the obfuscated *netperf* program and report the execution time and memory usage in Table VII. More specifically, we present the time and memory usage of the main components in each tool. The total analysis of Gadget-Planner takes around 100 minutes. The most time-consuming component is the gadget identification procedure, which is mainly related to the number of branches inside a program. Subsumption testing runs symbolic execution and constraint solving on each gadget, so its execution time depends on the complexity and number of the gadgets. Surprisingly, the planning component takes the least time. The main reason is that the two previous procedures have effectively reduced the search space. Empirically, they reduce the set of gadgets by an average factor of 2.97, which yields a gadget collection with a tractable size for the planning stage. Angrop is the fastest because it only focuses on hard-coded patterns. SGC takes a similar time as Gadget-Planner, where the most time and memory-consuming part is the gadget chaining using program synthesis. Overall, the performance of Gadget-Planner is competitive, especially considering the complexity and diversity chains it can build.

VII. Related Work

Planning has been used for program synthesis, although usually at a relatively high level and not to our knowledge in the context of code-reuse attacks. For example, Bhansali [62] uses hierarchical planning and analogical reasoning to generate simple Unix shell scripts from high-level specifications. Ireland and Stark [63] use proof planning and partial-order planning to generate correct imperative programs from specifications. Planning has also been used to automatically synthesize workflows of web services [64]. In the context of computer security, planning has been used to generate attacks on models of computer systems and networks as an aid in penetration testing [65], [66]. None of these works focus on code at the assembler level or on code-reuse attacks.

Ropper [67], RP++ [68], and Ropc [69] implement very similar gadget-finding functionality as ROPGadget, but less powerful, so we use ROPGadget in this work. BOPC [29] builds code-reuse attacks following CFI requirements. We did not compare with it because CFI does not work well with obfuscated programs. P-shape [70] has a very detailed paper but no available code.

VIII. Conclusion

Software obfuscation techniques hide program logic by adding complex data or controlling structure flow. Most existing works only focus on reversing the obfuscation but neglect the potential harm introduced by the obfuscation techniques. To answer this question, we first study the code-reuse vulnerability introduced by obfuscation methods and the result shows that it introduces a significant number of gadgets. However, we also reveal that existing tools cannot effectively construct code-reuse attacks due to the limited patterns, searching strategy, and complex gadgets from obfuscated programs. Hence we further propose a new technique to search code-reuse gadgets and payloads by leveraging the strength of symbolic execution and partial-order planning. We have implemented this method as a prototype called Gadget-Planner and evaluated it with popular obfuscators on benchmark programs and real-world open-source software. Compared with existing peer tools, Gadget-Planner can find much more code-reuse chains in obfuscated programs, demonstrating the potential risks introduced by software obfuscation.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. This research was supported by NSF grant CNS-1948489, CNS-2022279, and CNS-2211905.

References

[1] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, 2015.

[2] C. Collberg, "The Tigress C Obfuscator," https://tigress.wtf, [online].

[3] Oreans Technologies, "Code Virtualizer: Total Obfuscation against Reverse Engineering," http://oreans.com/codevirtualizer.php, [online].

[4] VMProtect Software, "VMProtect software protection," http://vmpsoft.com, [online].

[5] Oreans Technologies, "Themida: Advanced Windows Software Protection System," https://www.oreans.com/themida.php, [online].

[6] Stunnix, "Stunnix," http://stunnix.com/about/customers.shtml, 2022.

[7] R. Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT'09)*, 2009.

[8] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[9] J. Kinder, "Towards Static Analysis of Virtualization-Obfuscated Binaries," in *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*, 2012.

[10] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A Generic Approach to Automatic Deobfuscation of Executable Code," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[11] D. Xu, J. Ming, Y. Fu, and D. Wu, "VMHunt: A Verifiable Approach to Partial-Virtualized Binary Code Simplification," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[12] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–37, 2016.

[13] P. Wang, S. Wang, J. Ming, Y. Jiang, and D. Wu, "Translingual Obfuscation," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (Euro S&P'16)*, 2016.

[14] H. P. Joshi, A. Dhanasekaran, and R. Dutta, "Trading off a vulnerability: does software obfuscation increase the risk of rop attacks," *Journal of Cyber Security and Mobility*, pp. 305–324, 2015.

[15] ——, "Impact of software obfuscation on susceptibility to return-oriented programming attacks," in *36th IEEE Sarnoff Symposium*, 2015, pp. 161–166.

[16] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," The University of Auckland, Tech. Rep., 1997.

[17] 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.

[18] D. Xu, J. Ming, and D. Wu, "Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method," in *Proceedings of the 19th Information Security Conference (ISC'16)*, 2016.

[19] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 2009.

[20] R. Manikyam, J. T. McDonald, W. R. Mahoney, T. R. Andel, and S. H. Russ, "Comparing the Effectiveness of Commercial Obfuscators Against MATE Attacks," in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW'16)*, 2016.

[21] M. Polychronakis, *Reverse Engineering of Malware Emulators*. Springer US, 2011, ch. Encyclopedia of Cryptography and Security.

[22] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[23] 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*, 2007.

[24] 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)*, 2011.

[25] 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*, 2010.

[26] A. Francillion and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, 2008.

[27] 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.

[28] 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 *IEEE Symposium on Security and Privacy (SP)*, 2016.

[29] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[30] J. Salwan, "Ropgadget," http://shell-storm.org/project/ROPgadget, 2011. [Online]. Available: http://shell-storm.org/project/ROPgadget/

[31] A. team, "Angrop – a rop gadget finder and chain builder," https://github.com/angr/angrop, 2021.

[32] M. Schloegel, T. Blazytko, J. Basler, F. Hemmer, and T. Holz, "Towards automating code-reuse attacks using synthesized gadget chains," in *European Symposium on Research in Computer Security*, 2021.

[33] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[34] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, 2014.

[35] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012.

[36] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015.

[37] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, 2017.

[38] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, 2017.

[39] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[40] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[41] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh, "Cryptographically enforced control flow integrity," 2014.

[42] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Almgren, V. Gulisano, and F. Maggi, Eds., 2015.

[43] V. Pappas, "kbouncer: Efficient and transparent rop mitigation," http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf, 2012.

[44] S. Sinnadurai, Q. Zhao, and W. Fai Wong, "Transparent runtime shadow stack: Protection against malicious return address modifications," https://zatoichi-engineer.github.io/assets/docs/10.1.1.120.5702.pdf, 2008.

[45] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: a detection tool to defend against return-oriented programming attacks," in *AsiaCCS*, 2011.

[46] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[47] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE Symposium on Security and Privacy*, 2014.

[48] 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 *The 23rd USENIX Security Symposium*, 2014.

[49] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address-oblivious code reuse: On the effectiveness of leakage-resilient diversity," in *NDSS Symposium 2017*, 2017.

[50] K. Z. 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 *2013 IEEE Symposium on Security and Privacy*, 2013.

[51] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, 2009.

[52] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 IEEE Symposium on Security*, 2013.

[53] S. Banescu, C. Collberg, and A. Pretschner, "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.

[54] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Secruity and Privacy*, 2016.

[55] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson, 2020.

[56] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[57] E. Karpas and D. Magazzeni, "Automated planning for robotics," *Annual Review of Control, Robotics, and Autonomous Systems*, 2020.

[58] S. Edelkamp and S. Schrödl, *Heuristic Search: Theory and Applications*. Morgan Kaufmann, 2011.

[59] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[60] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, 2006.

[61] R. Jones, "Netperf," https://github.com/HewlettPackard/netperf, 2021.

[62] S. Bhansali, "Domain-based program synthesis using planning and derivational analogy," *AI Magazine*, 1991.

[63] A. Ireland and J. Stark, "Combining proof plans with partial order planning for imperative program synthesis," *Automated Software Engineering*, 2006.

[64] P. Bertoli, M. Pistore, and P. Traverso, "Automated composition of web services via planning in asynchronous domains," *Artificial Intelligence*, 2010.

[65] M. Boddy, J. Gohde, T. Haigh, and S. Harp, "Course of action generation for cyber security using classical planning," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2005.

[66] J. Hoffmann, "Simulated penetration testing: From "dijkstra" to "turing test++"," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2015.

[67] S. Schirra, "Ropper," https://scoding.de/ropper, 2019. [Online]. Available: https://scoding.de/ropper/

[68] A. Souchet, "rp++," https://github.com/0vercl0k/rp/, 2017. [Online]. Available: https://github.com/0vercl0k/rp/

[69] Pakt, "ropc: A turing complete rop compiler," https://github.com/pakt/ropc, 2013. [Online]. Available: https://github.com/pakt/ropc

[70] A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden, "Pshape: Automatically combining gadgets for arbitrary method execution," in *Security and Trust Management*. Springer International Publishing, 2016.