Correctness and Flexibility in the Realm of Automated Code-Reuse Attacks

[*] or an Empirical Investigation of ROP-Chain Generators

Oskar Lund
Linköping University
Linköping, Sweden
osklu130@student.liu.se

Logan Shaffer
Linköping University
Linköping, Sweden
logsh641@student.liu.se

John Åkerman
Linköping University
Linköping, Sweden
johak760@student.liu.se

***Abstract*—Over the years ROP-chain generators have adopted several different methods in constructing ROP-chains including pattern matching, heuristics, and exploratory methods. This study investigates the current landscape of ROP-chain generators in terms of correctness and flexibility. Using ROP-benchmark we tested five publicly available ROP-chain generators including: *Ropgadget*, *Ropper*, *Angrop*, *Ropium*, and *SGC*. Each tool was measured ten times with ASLR enabled and disabled on seventeen representative binaries. Contrary to assumptions our findings show that next gen generators do not preform as well as you would suspect. While heurisitics-based generators such as *Ropium*, and *Angrop* successfully created valid chains for 75% and 11% respectively from binaries that contained a syscall instruction.**

***Index Terms*—Return oriented programming, empirical study, ROP-Chain generator.**

I. Introduction

Due to the widespread use of system-level languages, such as *C* and *C++*, which lack memory safety, binary exploitation has become a major concern in cybersecurity research. Since binaries directly interact with the operating system and system hardware, binary exploits pose a threat to various systems, including web servers and critical infrastructure. Binary exploitation techniques allow attackers to manipulate executing programs by injecting malicious code and altering their intended behavior. Over several generations of binary exploits, *Return-Oriented Programming* (ROP) has emerged as an effective method to bypass mitigations such as *Data Execution Prevention* (DEP). By leveraging existing program instructions, ROP redirects program execution into a chain of commands that, at worst, allows an attacker to achieve *Arbitrary Code Execution* (ACE).

Since its introduction over a decade ago, ROP exploits have become increasingly automated through the use of programs known as ROP-chain generators. These generators employ various techniques to create and implement ROP-chain attacks, including pattern matching [1], [2], heuristics [3], [4], *Satisfiability Modulo Theory* (SMT) solvers [5], and partial-order planning [6]. Due to the diversity of these approaches, multiple evaluation methods have been explored, including efforts by Schloegel [7] and Nurmukhametov [8]. While both studies compare the performance of their ROP-chain generators to other publicly available tools, neither provides an objective investigation into the correctness of the available ROP generators.

The structure of this thesis is as follows: In Section II, the background is presented, providing a foundation for the work conducted in this project. Section III discusses related work, focusing on ROP-chain generation and validation, and outlines the research questions. Next, the approach for addressing the research questions is detailed in section IV. Section V presents the results of our conducted tests, which are then discussed in Section VI, organized into parts based on the research questions.

II. Background

A. Buffer overflow exploitation

Due to the low level nature of languages such as assembly, *C*, and *C++*, automated memory/garbage collection is not present such as in high level languages such as *C#* and *Java*. This lack of automation increases performance drastically but, relies on the author to sanitize inputs and unallocate memory to maintain program security. An early exploitation example includes Stack buffer overflows. Where an attacker injects a string of charters much larger than expected by the program overwriting the return address, and redirect execution to their own code, typically injected into the stack or other memory regions under the attacker's control [9].

B. Buffer overflow mitigation

To address this memory vulnerability, security measures such as DEP and W$\oplus$X were implemented [10]. These techniques safeguard memory by marking regions as either writable or executable, but not both, preventing attackers from inserting code into writable areas and executing it. DEP refers to the mitigation implemented by *Windows*, while W$\oplus$X is deployed by *OpenBSD*, both rely on the same mechanism by utilizing the NX-bit in the *Central Processing Unit* (CPU), which marks pages as non-executable. In systems without NX-bit support, this behavior can be emulated.

C. Code reuse attacks

However, DEP or W⊕X alone is insufficient to prevent code-reuse attacks like ROP, as these attacks do not require the execution of code in pages marked with the NX-bit. Instead, they involve executing existing instructions from text segments or libraries such as *libc* [10]. A common example of such an attack is the *return-to-libc* attack, where attackers redirects execution to a legitimate function that should be inaccessible. Return-to-lib-c chains together available library instructions to deviate the intended control flow [11]. Other variations of these code reuse attacks include *Jump oriented Programming* (JOP) chaining together JMP instructions, and *Call Orientated Programming* (COP) using the call instructions.

D. Return oriented programming

ROP like other code reuse exploits was developed in response to various prevention techniques designed to hinder attacks that execute malicious code, often referred to as shellcode, after hijacking a program's control flow [10]. Instead ROP utilizes code that already exists in the program. ROP exploits sequences of legitimate instructions, known as gadgets, within the program's executable memory. These gadgets typically end with a ret (return) instruction, allowing attackers to chain multiple gadgets together to achieve their objectives. To execute a ROP attack, the attacker first diverts the program's control flow, often by overwriting a return address with the address of a gadget, rather than with shellcode. This approach circumvents protections like W⊕X and DEP, as it uses existing code rather than requiring the execution of newly inserted code.

E. ROP Mitigations

Control Flow Integrity (CFI) is a mitigation technique that theoretically prevents ROP attacks by ensuring that the program's control flow follows a predetermined plan or *Control Flow Graph* (CFG) [12]. Any deviation from this plan can be detected and stopped. The downside of this approach is the significant overhead it introduces, as well as the increase in binary size. However, ongoing research is exploring ways to reduce this overhead, making CFI more practical and less noticeable in terms of performance impact.

Another technique that makes it more difficult to set up a ROP attack is *Address Space Layout Randomization* (ASLR), which is used by the operating system to randomize the addresses of loaded code segments, such as programs and libraries [12], [13]. This makes it harder for an attacker to know where functions and instructions are located. In the case of a return-to-libc attack, for example, the attacker needs to know where *libc* is loaded. However, previous research shows that attackers can often leak information about the locations of code segments. For instance, in some cases the base address of the program image is not randomized, allowing attackers to leak information about library functions through the *Procedure Linkage Table* (PLT) [13].

F. ROP-chain generators

There are various ways to leverage ROP for program exploitation. While gadgets can be identified and assembled into chains manually, tools known as ROP-chain generators have begun automating this process. *Q*, one of the first ROP-Chain generators divides this process into three operations; gadget discovery, gadget arrangement, and gadget assignment [14]. While each tool accomplishes theses tasks uniquely the underlying process is similar. Tools begin by identifying gadgets from the targeted program's binary. Next, the selected gadgets are arranged into possible chains. Last, the chains validity is assessed according to the attackers desired goal. Common goals of ROP-chains include executing a system call to spawn a shell on the host system or marking a memory region as executable, allowing redirection of execution to shellcode. The techniques used in the synthesis of these chains vary among tools, and the following sections explore these different approaches.

G. Pattern matching

The simplest approach to automation of ROP-chains relies on hard coded regular expressions [7], [15]. Tools that fall into this category are e.g. *Ropper* [1] and *ROPGadget* [2]. ROP-chain generators in this category works in two stages, first it searches the target binary for gadgets and secondly attempts to chain gadgets together in a predefined way [15]. Relying on hard coded rules and the restriction of gadgets limits the flexibility of this type of ROP-chain generator, typically only supporting the most common chain i.e. an execve call to launch a shell on the target system [7].

H. Heuristics-based

ROP-chain generators that fall into this category tend to use dependency graphs and different search algorithms to locate suitable gadgets to form a chain [16]. Compared with hardcoded-based, heuristics-based can sometimes utilize more complex gadgets with side-effects e.g. writing to memory but still discard many gadgets that could be usable [16]. ROP-chain generators belonging to this category include *Angrop* [4] and *Ropium* [3].

I. Exploratory approach

1) Satisfiability Modulo Theory (SMT) solver: An exploratory approach that uses an SMT solver discovers exploitable gadget sequences by constructing the problem as a set of logical constraints and deciding the satisfiability of logical formulas. These types of solvers are particularly effective in symbolic execution, which is often applied to explore all possible execution paths in a program by treating inputs as symbolic values rather than concrete ones. This approach is put to use in the tool *SGC* [5]. The design of *SGC*, as described by Schloegel et al. [7], is based on establishing preconditions and postconditions that describe the initial state and the desired state of the CPU during the exploitation process. These preconditions are set to the state before the execution of a possible gadget chain which could include specific register values, memory states or set flags. The postconditions describe the expected state after the execution. Using a logical formula, the effect of each gadget is matched against the constraints required to transition from the preconditions to the postconditions, effectively generating a usable ROP-chain.

While SMT solvers are a powerful tool for systematically exploring potential ROP-chains, they also represent a significant performance bottleneck. As highlighted by Schloegel et al. [7], the solver may require considerable time to identify valid gadget chains.

2) Partial-order planning: Partial-order planning is an *Artificial Intelligence* (AI) technique used in *Gadget-Planner* [6] to find a sequence of gadgets that can be used to build a ROP-chain. Total-order planning works by building a directed tree graph where each node represents a specific system state and each edge indicates a specific action used to change the state of the system from one state to another. Partial-order plannings key difference is that the order of gadgets is only partially specified where order is only enforced on actions that are dependent on each other while non interfering gadgets position if flexible [15]. Because of that the state space becomes significantly smaller and faster to search. Parietal-order planning starts at the goal state and searches backward for gadgets to fulfill its preconditions.

III. Related work

A. ROP-chain generation

Several state-of-the-art ROP-chain generation tools have been developed, employing various techniques for generating chains from discovered gadgets. Some tools, such as *Angrop* [4] and *Ropium* [3], utilize pattern-based heuristic techniques to find chains that achieve predefined end goals. In contrast, exploratory approaches, such as the use of SMT solvers, are employed by tools like *SGC* [5], as presented by Schloegel et al. [7], and *TGRop* [17], as introduced by Zhong et al. [16]. Furthermore, Zhang et al. [15] demonstrated how ROP-chain generation can be achieved using an AI approach, employing partial-order planning in the tool *Gadget-Planner* [6].

B. ROP-chain validation

There are several options available for ROP-chain verification. One approach is to generate the ROP-chain on a binary containing a known vulnerability, as was done by Schloegel et al. in [7] with a vulnerable version of *dnsmasq*. Although verifying binaries through exploitation is as close to real life as possible, it limits the possible binaries for which the ROP-chains can be verified. To avoid this limitation, a programmable debugger such as *gdb* can be used to insert the ROP-chain into the memory of the running binary and redirect execution to the beginning of the chain [7]. Another approach for verification was created by Nurmukhametov et al. in [8], called *rop-benchmark* [18]. *rop-benchmark* provides a test suite, with multiple popular ROP-chain generators, which tests the tools ability to generate a working execve chain to launch a shell [8]. To facilitate the verification of the chains, *rop-benchmark* embeds the target binary in the vulnerable program such that the chain can be executed [18]. Zhong et al. [16] further extends *rop-benchmark* to include *SGC* [5] as well. Using the following criteria, we will discuss the flexibility and correctness of these ROP-chain generators.

- **RQ1:** What are the procedures to verify a generated ROP-chain? *(correctness)*
- **RQ2:** How does the flexibility of selected ROP-chain generators vary in constructing different types of exploits? *(flexibility)*
- **RQ3:** How does the false positive rate differ among the selected ROP-chain generators in constructing functional exploit chains? *(correctness)*
- **RQ4:** How does the exploitation flexibility of the ROP-Chain generator impact its ability to output a valid ROP-chain? *(flexibility)*

IV. Approach

A. Selecting ROP-chain generators

The ROP-chain generators were selected to create a diverse and representative subset of available tools (supporting x86-64 Linux binaries). Diversity of software could be quantified in a multitude of ways, as such two aspects of the ROP-chain generators were selected for this purpose: origin and category. The origin of each ROP-chain generator was determined to be either academic or non-academic. Non-academic does not include proprietary ROP-chain generators, since source code and documentation availability were deemed a necessity for testing purposes. *Github* [19] was used to find open-source candidates. The categorization presented by Zhong et al. in [16] was used for the categorization in this project, due to it being quite recent and comprehensive. This categorization includes three categories: Pattern matching approaches, Heuristics-based approaches and exploratory approaches. At least one ROP-chain generator per category should be selected to have a diverse enough set.

Furthermore, ROP-chain generators in the exploratory category should use different strategies, e.g. two ROP-chain generators utilizing an SMT-solver would reduce the diversity. For the selection of ROP-chain generators to be representative, actively used and maintained projects should be preferred. By inspecting repository meta-data: Last commit, number of contributors, stars, forks, watchers and dependents and documentation quality the level of representativeness could be used to gauge repository activity. The selected ROP-chain generators are listed in Table I.

TABLE I
Selected ROP-Chain Generators

Name	Category	Academic
Gadget-Planner	Exploratory	yes
ROPGadget	Pattern matching	no
Ropper	Pattern matching	no
Angrop	Heuristics	no
SGC	Exploratory	yes
Ropium	Heuristics	no

B. *Selecting binary files*

The selection of binary files was based on a combination of file size, executable content and functionality. The goal was to select a diverse set of binaries in order to evaluate the performance of ROP-chain generators across different types of binaries. A requirement for the selection was that all binaries had to be compiled for the x86-64 architecture to guarantee compatibility with the test environment.

The criteria for the binary files were as follows:

- **File size variability**: Binaries were selected to represent a wide range of sizes, from small utilities to large libraries and server software. The goal was to understand how ROP-chain generators perform when handling binaries with differing complexity. The range of the size was determined to be between 20 kilobytes (kB) and 12 megabytes (MB).
- **Functional categories**: The selection aimed to cover a variety of functional domains, including networking tools, cryptographic libraries, file management utilities, process and memory management and system libraries.

The share of executable code within each binary was considered important, as this can influence how much of the binary is involved in ROP-chain generation, potentially affecting the number of gadgets and chains generated. The size of the executable code was determined by executing the `readelf -s <binary>` command on each binary and summarizing the sections with executable privileges. Additionally, to ensure the reproducibility of the study, the version of each binary was documented. This allowed the same versions to be used across tests. The selected binaries can be seen in Table II.

C. *Test Environment*

The test environment was setup to facilitate test execution, while ensuring reproducibility of the results. Therefore, the test environment was based on the *rop-benchmark* [18] tool developed by Nurmukhametov et al. in [8]. The *rop-benchmark* tool was also used by Zhong et al. in [16], which indicates some level of usefulness and reliability. The repository was forked to allow customizations to the benchmark tool, available on Github [20].

The modifications and additions required in the *rop-benchmark* tool were essentially to add support for the tools that were missing from the *rop-benchmark* and to include our binary selection. Since the *rop-benchmark* repository has not seen any code updates since August 2020, another required modification was to the *Dockerfile* such that the latest version of all ROP-chain generators was installed in the test environment. While updating all tools in the docker image, all the tools were installed in their own Python virtual environment to avoid version mismatch for shared dependencies.

The *rop-benchmark* tool supported all our selected ROP-chain generators except the academic ones, i.e. *SGC* and *Gadget-Planner*. Therefore, only modifications related to the Python virtual environment and ROP-chain generator output parsing were required for the already supported tools. The *SGC* scripts that Zhong et al. added to their copy of the *rop-benchmark* [17] tool in [16] were used as a starting point. The scripts were then adapted to work in our version of the *rop-benchmark*. *Gadget-Planner* was added to the docker image, and it can be run directly inside the docker container. However, multiple issues were found that prevented us from creating runner scripts for it. The lack of documentation, poor code quality, and long runtime of the tool led to its exclusion from the tests.

D. *ROP-chain verification*

All selected ROP-chain generators, listed in Table I, support the *execve "/bin/sh"* ROP-chain. The *rop-benchmark* tool [18] has a verification method included for *execve "/bin/sh"* ROP-chains, which we decided to use as well. There are two parts to the verification procedure: The addition of a stack-based buffer overflow vulnerability for the target binary and a feedback mechanism if */bin/sh* was launched. To add a vulnerability to the target binary, there exists a *Makefile* in *rop-benchmark* that will link the target binary to a small program with a known vulnerability in it. Then after a ROP-chain generator has produced a ROP-chain for that particular binary, it can be used as an argument to the vulnerable version of the binary such that it is written to the stack of the program. In order to determine whether the ROP-chain managed to launch */bin/sh*, a symlink has been created from */bin/sh* pointing to a script that simply echoes *SUCCESS* to indicate that the ROP-chain worked as intended.

E. *Benchmark Tests*

To evaluate the validity of the chains, all ROP-chain generators were configured to attempt to find a ROP-chain that executes *execve("/bin/sh")*. Each tool was run 10 times for each binary to account for variations. The benchmark outputs were then compiled into presentable results, capturing the valid chains, invalid chains, failed chain generations, and timeouts.

F. *Exploitation flexibility*

Based on previous works by Zhong et al. in [16], Nurmukhametov et al. in [8], and Schloegel et al. in [7], we have a fundamental understanding of the limitations of each selected ROP-chain generator. However, we will analyze the source code and documentation of each ROP-chain generator to further understand what exploits it supports. In addition to understanding which exploits each tool supports, we will also determine how the different exploits are configured in each ROP-chain generator. In the code analysis we will only consider configuration options that are directly related to different exploitation techniques. For instance, options to filter out gadgets containing certain bytes will not be evaluated during this code analysis.

G. *Limitations*

The benchmark tests were conducted on two available PCs the first 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz, 12GB RAM @26667 MHz , and a second PC with

TABLE II
THE CHOSEN BINARIES WITH SHARES OF HOW MUCH CODE HAS EXECUTABLE PRIVILEGES.

Binary	Version	Size (kB)	Executable section size (kB)	Share of executable	Category
/usr/bin/gdb	Gdb 15.0.50.20240403-0ubuntu1	11744	7045	61.42%	Debugger
/usr/Python3.12	Python3 3.12.3-0ubuntu2	8019	2955	37.73%	Library
/lib/x86_64-linux-gnu/libc.so.6	Libc6 2.39-0ubuntu8.3	2125	1567	75.49%	Library
/usr/bin/bash	GNU bash, version 5.2.21(1)-release	1446	953	66.63%	Shell utility
/usr/bin/openssl	Openssl 3.0.13-0ubuntu3.4	982	464	47.26%	Cryptography
/usr/sbin/sshd	Openssh-server 1:9.6p1-3ubuntu13.5	917	578	64.48%	Server utility
/usr/bin/wget	1.21.4-1ubuntu4.1	470	279	60.72%	Network
/bin/tar	Tar 1.35+dfsg-3build1	432	293	69.33%	File Management
/bin/netstat	Net-tools 2.10-0.1ubuntu4	158	77	49.47%	Network
/bin/ls	Coreutils 9.4-3ubuntu6	142	83	60.01%	File listing
/bin/touch	Coreutil 9.4-3ubuntu6	96.8	56	59.02%	File Management
/bin/ping	ping from iputils 20240117	89	43	48.73%	Network
/sbin/ifconfig	Net-tools 2.10-0.1ubuntu4	79	43	54.62%	Network
/bin/sleep	Coreutils 9.4-3ubuntu6	35	13	38.42%	General utilities
/bin/free	procps-ng 4.0.4	27	7	28.57%	Memory Management
/bin/kill	procps-ng 4.0.4	23	4	17.76%	Process Management
/bin/clear	6.4+20240113-1ubuntu2	15	1.4	9.13%	Shell utility

12th Gen Intel(R) Core(TM) i5-12400F @2.50GHz, 16GB RAM @3200MT/s. Because of the implementation of ROP-benchmark each generator intense is run single threaded. When comparing runs times between the computers we found no differences in timeouts. However, Some tools may require additional memory and computational power to function optimally.

V. RESULTS

In this section, the results from the test runs, as well as the findings from the qualitative analysis of each ROP-chain generator's flexibility, are presented.

A. ROP-chain Generation

The results from *rop-benchmark* are presented in the format *valid chain/invalid chain/failed/timeout*, as shown in Tables III and IV.

B. ROP-chain generator exploitation flexibility

Here we list the supported exploits of all ROP-chain generators, as well as how different exploits are configured.

ROPGadget

- execve("/bin/sh") - Command line flag *–ropchain*

Ropper

- `execve(cmd)` - Command line flag argument *–chain execve*. The `cmd` is the pathname supplied to `execve`, for example, *–chain "execve=/bin/ping"*. The default value for `cmd` is `/bin/sh`.
- `mprotect(address, size)` - Command line flag argument *–chain "mprotect address=0xbfdff000 size=0x21000"*. Both `address` and `size` are required arguments. The `address` must be aligned to a page boundary[1]. This sets the permissions to read, write, and execute.

Ropium

- ROP-chains are created using *Ropium* queries (Semantic queries). These semantic queries support: writing/reading registers, writing/reading memory, executing function calls with arguments (if function address is known), executing syscalls (by name or syscall number).
- Semantic queries are created using the Command Line Interface of *Ropium*, or in a Python script using the Python API.
- syscalls supported by name in Semantic queries are found in *ropium/libropium/compiler/systems.cpp*.

Angrop

- ROP-chains are created using the methods, except for *__init__* in the ChainBuilder class (angrop/chain_builder/__init__.py).
- Provides methods for executing function calls, syscalls, writing to registers, moving values between registers, writing to memory, addition to value in memory.
- Provides a convenience method to build an execve ROP-chain.

SGC - Gadget Synthesis

SGC was the tool requiring the most configuration before it could run. To execute it, two JSON configuration files needed to be prepared[2]:

1) `synthesizer_config_default.json`
2) `config_execve.json`

To enable *SGC* for the benchmark, the implementation was based on the TGrop [17] integration of *SGC*. The synthesizer configuration file was set up with the following parameters (the most relevant):

- `"block_limits": [100,300]` Defines the size of the sampled gadget subset.

[1]See the mprotect manual at https://www.man7.org/linux/man-pages/man2/mprotect.2.html

[2]See the https://github.com/RUB-SysSec/gadget_synthesis/tree/master/targets file for more details.

TABLE III
COMPARISON OF ROP-CHAIN GENERATORS, TABLE CELL FORMAT: VALID CHAIN / INVALID CHAIN / FAILED / TIMEOUT WITH ASLR ENABLED

Binary	ROPGadget	Ropper	Angrop	Ropium	SGC
gdb	0 / 10 / 0 / 0	0 / 10 / 0 / 0	0 / 0 / 0 / 10	9 / 1 / 0 / 0	- / - / - / -
Python3.12	0 / 10 / 0 / 0	0 / 10 / 0 / 0	0 / 0 / 0 / 10	10 / 0 / 0 / 0	- / - / - / -
libc.so.6	0 / 10 / 0 / 0	0 /10 / 0 / 0	0 / 0 / 0 / 10	10 / 0 / 0 / 0	- / - / - / -
bash	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 10 / 0 / 0	- / - / - / -
openssl	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -
sshd	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 9 / 1	10 / 0 / 0 / 0	- / - / - / -
wget	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	10 / 0 / 0 / 0	- / - / - / -
tar	0 / 0 / 10 / 0	0 / 10 / 0 / 0	10 / 0 / 0 / 0	10 / 0 / 0 / 0	- / - / - / -
netstat	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -
ls	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -
touch	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -
ping	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -
ifconfig	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -
sleep	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -
free	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -
kill	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -
clear	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	- / - / - / -

TABLE IV
COMPARISON OF ROP-CHAIN GENERATORS, TABLE CELL FORMAT: VALID CHAIN / INVALID CHAIN / FAILED / TIMEOUT WITH ASLR DISABLED

Binary	ROPGadget	Ropper	Angrop	Ropium	SGC
gdb	0 / 10 / 0 / 0	0 / 10 / 0 / 0	0 / 0 / 0 / 10	10 / 0 / 0 / 0	0 / 0 / 1 / 9
Python3.12	0 / 10 / 0 / 0	0 / 10 / 0 / 0	0 / 0 / 0 / 10	10 / 0 / 0 / 0	0 / 0 / 0 / 10
libc.so.6	0 / 10 / 0 / 0	0 /10 / 0 / 0	0 / 0 / 0 / 10	10 / 0 / 0 / 0	0 / 0 / 0 / 10
bash	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 0 / 10
openssl	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 8 / 2 / 0
sshd	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	10 / 0 / 0 / 0	0 / 0 / 1 / 9
wget	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	10 / 0 / 0 / 0	0 / 0 / 0 / 10
tar	0 / 0 / 10 / 0	0 / 10 / 0 / 0	10 / 0 / 0 / 0	10 / 0 / 0 / 0	0 / 0 / 0 / 10
netstat	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0
ls	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0
touch	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0
ping	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0
ifconfig	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0
sleep	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0
free	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0
kill	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0
clear	0 / 0 / 10 / 0	0 / 10 / 0 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0	0 / 0 / 10 / 0

- `"disassemble_unaligned": false` Indicates that gadgets must be aligned, meaning they must consist of complete x86-64 instructions.
- `"control_flow_types": ["ret","call","jmp"]` Specifies the types of gadgets used in the synthesis, including ROP, COP, and JOP.
- `"selection_strategy": "seed"` Determines how the gadget subset is sampled. In this configuration, a seeded pseudo-random selection strategy was used.
- `"initial_seed": 0` The initial seed value for the selection strategy.
- `"solver_timeout": 3600` Sets the timeout (in seconds) for the SMT solver.

These configuration fields were chosen based on the TGrop implementation [17] and the example configurations for pre-configured targets in the *SGC* repository [5].

The second configuration file, `config_execve.json`, had to be adjusted for each binary. This was achieved by using a pre-configured set of fields that remained static across all runs. Some fields, however, required specific configuration for each binary, including the following:

- `read_mem_areas` A list of ranges representing readable memory areas. The addresses of the readable memory regions were extracted for each binary using *gdb* in the runner script for SGC in *rop-benchmark*.
- `write_mem_areas` A combination of pre-configured stack ranges and dynamically added ranges for the `.bss` section.
- `"postconditions": [["IRDst", "", 64], ...]` Specifies the desired address of the next instruction to be executed at the end of the ROP-chain. This should be the address of a syscall instruction if used for an *excve* exploit.

The values for these fields were determined using a combination of tools such as *GDB*, *ROPgadget*, and *objdump*. The full configuration templates used in the testing process are provided in Appendices A and B.

VI. Discussion

In this section we will attempt to answer each of our posed research questions based on the acquired results. Furthermore, we will discuss potential shortcomings of our test setup, verification method, the ROP-chain generators, as well as our code analysis. We will also discuss future work.

A. *RQ1 - What are the procedures to verify a generated ROP-chain?*

There are basically three steps required to verify a ROP-chain using the target binary: inserting the ROP-chain into executable memory in the binary, redirecting execution to the first gadget in the ROP-chain, and lastly checking for the symptoms of execution. As described in Section III-B, inserting the ROP-chain into executable memory can be accomplished using a real or synthetic buffer overflow vulnerability in the target binary. Otherwise a programmable debugger, or some other tool, that allows for modification of the executable memory in a binary, can be used to insert the ROP-chain. Redirecting execution is simply a matter of changing the program counter register with a programmable debugger, or overwriting the return address correctly when using buffer overflow vulnerabilities for insertion. The two first steps remain constant for all types of exploits contained within a ROP-chain. However, the symptoms of exploitation on a system will depend on type of exploit and it requires a lot more knowledge of the exploitation to design a simple method of checking that it was successful or not. The *execve("/bin/sh")* ROP-chain has known symptoms and is easy to verify, in addition to the general usefulness of this exploit this might be a reason for its popularity in other papers evaluating ROP-chain generators [7], [8], [15]. Schloegel et al. [7] validates ROP-chains for a *mprotect* call as well, however it is not clear exactly how they conduct the validation. If the buffer overflow approach is used to insert the ROP-chain, it can also insert an *execve* payload as well. Then as long as the stack does not have executable permissions set initially, the ROP-chain can be validated using this method by observing if the exploit produce a segmentation fault or not. However, that seems quite error prone although realistic and a better method is probably to check the permissions for the target memory page in the kernel (i.e., /proc/<PID>/maps).

Since we opted to use *rop-benchmark*, the insertion and execution redirection were accomplished through a synthetic buffer overflow. By running the target binary with the ROP-chain as an argument, the ROP-chain is supposed to be put on the stack and the return address of the vulnerable function replace by the address of the first gadget. Inputing the ROP-chain in this manner requires it to be a valid string in C, since that is the language used to write the vulnerable function. Strings in C are null-terminated i.e. ending with '\0'. Thus if the ROP-chain contains such characters, the entire chain will not be inserted into memory and most likely fail to run the exploit. This is not a problem when using a programmable debugger to insert the ROP-chain, since you can just write the byte sequences as numbers which allows for null-characters. All of the selected ROP-chain generators for this project have options to remove particular bytes from the ROP-chain, essentially limiting their gadget search space. Schloegel et al. utilized this for their first run experiment in [7]. We did not consider this when setting up our tests for this project. Since we did not instruct the tools of the limitations of our validation procedure, some of the false positives in the result may be attributed to this decision. However, no null bytes were found in any of the generated ROP-chains.

B. *RQ2 - How does the flexibility of selected ROP-chain generators vary in constructing different types of exploits?*

Based on our evaluation of the flexibility of each ROP-chain generator in Section V-B we can determine the flexibility of each tool. *SGC*, *Angrop* and *Ropium* all have similar levels of flexibility, supporting most if not all possible exploits. *SGC* distinguishes itself from the other two, through the comprehensive configuration that enables the user to set the desired register and memory contents after ROP-chain execution. *Angrop* and *Ropium* offer similar levels of control to the user, but through an application programming interface (API). Both APIs enable the user to read/write to memory and registers, which in theory offers the same level of exploitation flexibility. However, depending on the actual implementation of the respective APIs, *Angrop* and *Ropium* might be slightly less flexible compared to *SGC*. *ROPGadget* and *Ropper* are both highly inflexible, only providing the user the ability to use exploits explicitly supported by each tool. *ROPGadget* only has support for *execve("/bin/sh")*, essentially offering no exploitation flexibility. *Ropper* is slightly more flexible, allowing the user to decide which argument to supply execve with. In addition to execve, *Ropper* also supports mprotect ROP-chains.

C. *RQ3 - How does the false positive rate differ among the selected ROP-chain generators in constructing functional exploit chains?*

From the results in Table IV and III, we can see that there are only minor differences between the results with ASLR enabled and disabled. The only differences are in *Angrop* and *Ropium*. A single test run for *Angrop* resulted in a timeout instead of a failed chain for the *sshd* binary with ASLR enabled. Similarly, there is a single difference in the *Ropium* result for *gdb* in one test run, where the chain was valid when ASLR was disabled instead of invalid when ASLR was enabled.

One possible explanation is that *Ropium* constructs the *gdb* ROP-chain with gadgets that contain constant memory addresses. The valid addresses of different sections in the binary will be randomized with ASLR enabled, such that a constant memory address might be valid for one run but invalid for another. Then by disabling the randomization, the addresses will either always be valid or invalid depending on how well the ROP-chain generator used this type of gadget.

Comparing the false positive rates between the tools and the binaries, there is a pattern of smaller binaries not resulting in any valid chains. All binaries smaller than the *tar* binary do not result in any valid chain. *Ropper* stands out in this range of binaries, as it gives 100% false positives, which also applies to all binaries. The other tools have all failed to generate a chain, which is reasonable as these binaries lack the syscall, as can be seen in Table VI. The exception in this range is the *ping* binary, which has an unaligned syscall gadget (gadgets with a partial x86-64 instruction containing 0x0f05) encoded in a *lea* x86-64 instruction. Some of the tools have the option of utilizing unaligned syscall gadgets and some do not. *Ropium* and *Angrop* have this functionality but still fail to make a chain for *ping* which has to do with the available gadgets. *SGC* also have this option of using unaligned gadgets but the authors commented that this option is not yet "fleshed out" [5] and they recommend to have it disabled. *Ropgadget* also has the option of making an offset into gadgets but does this for all gadgets.

Ropper is incapable of determining whether it has successfully constructed a ROP-chain and generates an output regardless of if it has generated a chain or not. We see a similar behavior in [8], though with a smaller binary set we are able to correct the output manually as seen in Table V. All empty ROP-chains and those only containing a syscall gadget were considered failed instead of invalid. Ropper test runs with ASLR enabled and disabled produced empty ROP-chains for the same binaries, as such ASLR information is omitted from Table V.

TABLE V
ROPPER OUTPUT FIXED, TABLE CELL FORMAT: VALID CHAIN / INVALID CHAIN / FAILED / TIMEOUT

Binary	Ropper
gdb	0 / 10 / 0 / 0
Python3.12	0 / 10 / 0 / 0
libc.so.6	0 / 10 / 0 / 0
bash	0 / 10 / 0 / 0
openssl	0 / 0 / 10 / 0
sshd	0 / 0 / 10 / 0
wget	0 / 0 / 10 / 0
tar	0 / 10 / 0 / 0
netstat	0 / 0 / 10 / 0
ls	0 / 0 / 10 / 0
touch	0 / 0 / 10 / 0
ping	0 / 0 / 10 / 0
ifconfig	0 / 0 / 10 / 0
sleep	0 / 0 / 10 / 0
free	0 / 0 / 10 / 0
kill	0 / 0 / 10 / 0
clear	0 / 0 / 10 / 0

In the binaries that is larger than *netstat* there is a available syscall for the tools. All these binaries have unaligned syscall gadgets except *gdb, Python3.12* and *libc.so.6* which have an explicit syscall instruction.

1) Ropium: Ropium finds valid chains for all of the binaries with explicit syscall and also finds valid chains for *tar*, *wget* and *sshd* but fails for *openssl* and generates a invalid chain for *bash*.

2) Angrop: Angrop only builds valid chains for *tar* with an unaligned syscall and either fails or timeouts for the remainder of the binaries. For many of the failed binaries *Angrop* was able to locate the unaligned syscall gadget, but was unable to generate a chain leading to execve. For the three binaries over 2 MB *gdb, Python3.12* and *libc.so.6* due to our testing benchmark and hardware constraints *Angrop* was unable to complete the gadget search phase before reaching the timeout threshold. If time constraints were not imposed *Angrop* it created a valid chain for each of the three binaries in a singular additional run with a timeout of 10000.

3) SGC: SGC is making no valid chains and mostly timeouts for the binaries with the syscalls. This differs from other tests conducted in [7], [15], which show that *SGC* performs better than *ROPGadget*, *Angrop*, and *Ropium* in finding chains on different programs, test cases, and benchmarks. However, tests similar to ours were conducted for the tool *TGRop* [16], which also observed that *SGC* often reach timeout and generates fewer valid chains compared to tested tools such as *Ropper*, *Ropium*, and *Angrop*. Notably, *ROPGadget* was the only tool that performed worse than *SGC* in the *TGRop* study [16]. We believe that there are several reasons why *SGC* does not perform as expected:

1) The time required by the SMT solver.
2) The complicated configuration needed.
3) The non-deterministic gadget selection.

Firstly, *SGC* often reaches a timeout due to the solver's time requirements. Based on the timeouts observed in the tests for the binaries, we note that binaries with an executable section larger than 293 kB (e.g., `tar`) fail to complete most of the tests. There is one exception: `openssl`, which consistently finishes but generates an invalid chain in eight out of ten test attempts. We can only speculate that this discrepancy occurs because the solver finds a "valid" chain early during the solving process.

Secondly, the many configuration options for *SGC* make the configuration process complex. There are two main configuration files: one configures the solver, and the other is used for configuring the specific binary to be used in the gadget discovery and chain synthesis. The solver configuration was set with the default settings from the GitHub repository, along with some specific options that were considered to be relevant for the solver performance. These options include the sampled gadget size, solver timeout, and the selection strategy for the gadgets (e.g., pseudorandom, deterministic, or pseudorandom with a seed).

The default sampled gadget size was 100 and 300 gadgets, which were chosen for testing. In the *SGC* paper [7], it is mentioned that due to the time required by the solver, a subset of gadgets must be selected, which may result in not finding all possible chains (hence the gadget sample size). In our tests, we used the same seed for all trials, and there is a possibility that the gadget sample size is too small to be effective for the chosen binaries. If we were to increase the sample size, the solver would take even more time and would not scale well on our test machines, making larger sample sizes unfeasible for most test benches under the time constraints.

Furthermore, the configuration file for the binary has additional options, which can be difficult to set correctly, especially if the configuration needs to be automated, as in the benchmark. The most important configuration fields for our test setup were the preconditions, postconditions, readable and writable memory areas. The preconditions were a bit tricky to set correctly, as they included the RSP register which was required to be set.

This register was a topic of discussion, as its value changes depending on the actual payload used. We also observed that it changes from run to run, making it difficult to determine its importance. In some pre-testing, we noticed that *SGC* could find valid chains in some binaries outside of our binary selection, even if the RSP did not correlate exactly with the actual register value at runtime. Due to the complexities of selecting this register, a static RSP value was set and used for all binaries and tests. When comparing with the implementation in the testing for *TGRop* [17], we found that they followed the same approach.

The readable and writable memory areas also had to be set beforehand. These ranges are important because they determine where the solver can read and write memory in the ROP-chain. These configuration fields were also difficult to set correctly. During some pre-testing, we experimented with setting the writable areas to only allow the stack and also tested with a combination of the stack and other writable areas, such as the '.bss' section. The difference between these configurations was hard to assess as necessary or not, due to the non-deterministic nature of chain synthesis.

For the readable areas, we encountered a similar issue as with the writeable memory area. The configuration for this field was compared to the implementation in *TGRop* [17] and we decided to choose the full readable range of the '.text' section as the readable areas, which worked in some binaries outside of our binary selection. Whether the selection of the readable and writable areas affects the results is difficult to determine.

It can also be added that *SGC* itself has a validation functionality that checks the chains validity through symbolic execution. When the symbolic execution fails the found chain will be discarded which results in *failed* in Table IV. This also means that the invalid chains produced by *SGC* indicate that the *SGC* validation is either incorrect or does not match the CPU/memory states.

D. RQ4 - How does the exploitation flexibility of the ROP-Chain generator impact its ability to output a valid ROP-chain?

From the results of our tests, in Tables III, IV, and V we can see that the only tools that found valid ROP-chains were *Angrop* and *Ropium*. The fact that they were the only ROP-chain generators that found valid chains, is not necessarily a result of their exploitation flexibility. Since both *Angrop* and *Ropium* offer similar levels of exploitation flexibility as *SGC*, while *SGC* found no valid chains results indicate that high flexibility does not necessarily improve the tools ability to find valid ROP-chains. To actually find out if there is a correlation between exploitation flexibility and finding valid ROP-chains, more tests targeting other exploits than *execve("/bin/sh")* would have to be conducted. Since we have only tested for *execve("/bin/sh")* ROP-chains, the flexibility of the ROP-chain generators has not been tested. As such, our test results only support conclusions regarding the relation between different types of ROP-chain generators *(pattern matching, heuristics-based, exploratory)* and their ability to output valid ROP-chains.

TABLE VI
THE SYSCALL AVAILABILITY INCLUDING BOTH ACTUAL SYSCALLS AND UNALIGNED SYSCALL GADGETS, SUCH AS INSTRUCTIONS CONTAINING THE 0X0F05 SEQUENCE.

Binary Name	Has syscall
gdb.bin	yes
python3.12.bin	yes
libc.so.6.bin	yes
bash.bin	yes
openssl.bin	yes
sshd.bin	yes
wget.bin	yes
tar.bin	yes
netstat.bin	no
ls.bin	no
touch.bin	no
ping.bin	yes
ifconfig.bin	no
sleep.bin	no
free.bin	no
kill.bin	no
clear.bin	no

E. Future work

Several modern ROP-chain generators such as *Gadget-Planner, Majorca* and *TGRop* are not publicly available or have been released in a non-functional state. In future experiments the addition of these generators would aid in a more holistic understanding of modern ROP-chain generation, and how they compare to previous implementations. Another interesting research avenue is adding support for automated verification of ROP-chains for other exploits than *execve("/bin/sh")* into *rop-benchmark*. All selected ROP-chain generators, with the exception of ROPGadget, support generating *mprotect* ROP-chains. This makes *mprotect* a prime candidate as the next exploit to add automated ROP-chain verification for.

VII. Conclusion

In the current landscape of ROP-chain generators, heuristic based ROP generators outperform both pattern matching, and SMT based implementations in our restricted time frame. In binaries that contained a syscall instruction, *Ropium*, and *Angrop* successfully created executable chains for 75% and 11% respectively from this binary subset. While competing tools such as *ROPgadget, Ropper*, and *SGC* were unable to create a successful chain.

Acknowledgments

We would like to thank, our supervisor Ulf Kargén for his technical expertise in binary exploitation.

REFERENCES

[1] S. Schirra, “Ropper.” Available: https://github.com/sashs/Ropper. [Accessed: 2024-12-01].

[2] J. Salwan, “Ropgadget.” Available: https://github.com/JonathanSalwan/ROPgadget. [Accessed: 2024-12-01].

[3] Boyan-MILANOV, “Boyan-milanov/ropium.” Available: https://github.com/Boyan-MILANOV/ropium. [Accessed: 2024-12-01].

[4] angr, “angr/angrop.” Available: https://github.com/angr/angrop. [Accessed: 2024-12-01].

[5] RUB-SysSec, “Rub-syssec/gadget_synthesis.” Available: https://github.com/RUB-SysSec/gadget_synthesis. [Accessed: 2024-12-01].

[6] Softsec-Unh, “Softsec-unh/gadget-planner.” Available: https://github.com/softsec-unh/Gadget-Planner. [Accessed: 2024-12-01].

[7] M. Schloegel, T. Blazytko, J. Basler, F. Hemmer, and T. Holz, “Towards automating code-reuse attacks using synthesized gadget chains,” in *Computer Security – ESORICS 2021* (E. Bertino, H. Shulman, and M. Waidner, eds.), (Cham), pp. 218–239, Springer International Publishing, 2021.

[8] A. Nurmukhametov, A. Vishnyakov, V. Logunova, and S. Kurmangaleev, “MAJORCA: Multi-architecture jop and rop chain assembler,” in *2021 Ivannikov ISPRAS Open Conference (ISPRAS)*, pp. 37–46, IEEE, 2021.

[9] Aleph One, “Smashing the stack for fun and profit,” *Phrack*, vol. 7, November 1996.

[10] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Trans. Inf. Syst. Secur.*, vol. 15, Mar. 2012.

[11] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, “On the expressiveness of return-into-libc attacks,” in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID’11, (Berlin, Heidelberg), p. 121–141, Springer-Verlag, 2011.

[12] S. Das, W. Zhang, and Y. Liu, “A fine-grained control flow integrity approach against runtime memory attacks for embedded systems,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 11, pp. 3193–3207, 2016.

[13] A. V. Vishnyakov and A. R. Nurmukhametov, “Survey of methods for automated code-reuse exploit generation,” *Programming and Computer Software*, vol. 47, pp. 271–297, Jul 2021.

[14] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: Exploit hardening made easy,” in *20th USENIX Security Symposium (USENIX Security 11)*, (San Francisco, CA), USENIX Association, Aug. 2011.

[15] N. Zhang, D. Alden, D. Xu, S. Wang, T. Jaeger, and W. Ruml, “No free lunch: On the increased code reuse attack surface of obfuscated programs,” in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 313–326, 2023.

[16] N. Zhong, Y. Chen, Y. Zou, X. Xing, J. Dong, B. Xian, J. Zhao, M. Li, B. Liu, and W. Huo, “Tgrop: Top gun of return-oriented programming automation,” in *Computer Security – ESORICS 2024* (J. Garcia-Alfaro, R. Kozik, M. Choraś, and S. Katsikas, eds.), (Cham), pp. 130–152, Springer Nature Switzerland, 2024.

[17] ZoEplA, “Zoepla/tgrop.” Available: https://github.com/ZoEplA/TGRop. [Accessed: 2024-12-01].

[18] ispras, “ispras/rop-benchmark.” Available: https://github.com/ispras/rop-benchmark. [Accessed: 2024-12-01].

[19] GitHub, inc, “Github.” Available: https://github.com, 2024. [Accessed: 2024-10-13].

[20] CheddarJon, “Cheddarjon/rop-benchmark.” Available: https://github.com/CheddarJon/rop-benchmark. [Accessed: 2024-12-01].

Appendix A
SGC - synthesizer_config_default.json

```
{
    "all_iterations": [4],
    "solver": "boolector",
    "block_limits": [100,300],
    "restrict_mem": true,
    "disassemble_unaligned": false,
    "selection_strategy": "seed",
    "selection_location": "main_exe",
    "control_flow_types": ["ret","call","jmp"],
    "initial_seed": 0,
    "initial_block_offset": 0,
    "max_selection_variations": 2,
    "all_max_stack_words": [16],
    "solver_timeout": 3600,
    "disassembly_timeout": 3600
  }
```

Appendix B
SGC - config_execve.json

```
{
  "executable": "",
  "arch": "x86_64",
  "load_address" : "0x000000",
  "preconditions": [
                ["IRDst", "0x4013fa", 64],
                ["RSP", "0x7fffffffd518", 64]
            ],
  "postconditions": [
                ["IRDst", "", 64],
                ["RAX",  "0x3b", 64],
                ["RSI",  "0x0", 64],
                ["RDX",  "0x0", 64]
            ],
  "ptr_postconditions": [
            ["RDI", "/bin/fh", 64]
        ],
  "read_mem_areas": [],
  "write_mem_areas": [["0x7ffffffde000", "0x7ffffffff000"]]
}
```