An empirical comparison of open-source ROP-chain generators

Marcus Döberl
Linköping University
mardo435@student.liu.se

York Freiherr von Wangenheim
Linköping University
yorfr185@student.liu.se

Felix Strömberg Hooshidar
Linköping University
felst309@student.liu.se

Edvin Schölin
Linköping University
edvsc779@student.liu.se

***Abstract*—This study aims to compare and measure the differences between various open-source ROP-chain generation tools to better understand their capabilities, limitations, and efficiency across various binary file sizes and functionalities. The analysis focuses on the generators *angrop*, *Gadget-Planner*, *ROPgadget*, *ropium*, *Ropper*, and *Gadget Synthesis*. The methodology for this study is designed to empirically evaluate the capabilities of the selected generators when generating chains from various binary files of different sizes. The qualitative evaluation focused on identifying supported architectures and exploring additional code reuse attacks. The quantitative analysis revealed a polynomial relationship between the executable size of a binary and the time required by each generator to find the first chain. The study also revealed that the number of gadgets identified by the selected ROP generators generally increased with the size of the binary, confirming an polynomial relationship across all tools. The results of the qualitative evaluation highlight that most of the analyzed tools support x86-64, ARM, and MIPS. *Ropper* and *ROPgadget* extend compatibility further to PowerPC and Sparc, while *Gadget Synthesis* remains restricted to x86-64. Regarding attack types, all tools support ROP, while JOP is supported by all except *angrop*. Notably, *Gadget Synthesis* is the only tool observed to support COP, making it uniquely versatile for advanced code reuse attack scenarios.**

***Index Terms*—Return oriented programming, Code-reuse attacks, ROP-chain generators, gadgets**

I. Introduction

Arbitrary code execution exploits are among the most severe types of cyberattacks since they allow an attacker to implant and execute malicious code into a running process, sometimes allowing a complete takeover of the affected system. Return-oriented programming (ROP) has become a widely-used technique for exploiting memory vulnerabilities in programs, as it allows attackers to bypass defenses designed to prevent code injection. Instead of injecting new code, chains of small snippets of existing code within the target program are used, effectively circumventing the restrictions. For example, one such defense against code injection is *Write xor eXecute* (W$\oplus$X), which prevents memory pages from being writable and executable at the same time. However, ROP-attacks bypass W$\oplus$X by utilizing the program's executable code, forming chains of "gadgets" that are already marked as executable.

Since ROP-chains do not rely on injecting new code, they can avoid protections like W$\oplus$X, making it challenging for traditional defense mechanisms to detect and prevent such attacks. This has led to the widespread adoption of ROP in modern exploit techniques, highlighting the need for more advanced mitigation strategies that can counteract ROP attacks effectively. Consequently, analyzing and evaluating the efficiency of different ROP-chain generation tools has become critical in understanding how these attacks are constructed and how they can be defended against.

II. Background

A. Return-Oriented Programming

Code-reuse attacks occur when an attacker manipulates existing code to achieve a malicious outcome by controlling its flow[1]. These attacks pose a significant threat to the integrity and confidentiality of a system by being able to bypass standard security measures, sometimes even well-established ones[2]. One prominent example of a code reuse attack is Return-Oriented Programming (ROP).

Return-oriented programming works by an attacker stringing together short sequences of machine instructions, called gadgets, that already exist within the program's memory. These gadgets create a chain of instructions that redirects a program's intended control flow, allowing an attacker to execute arbitrary instructions instead [3].

A gadget is typically a small set of instructions that performs a simple operation such as moving data, arithmetic, logic and system calls. Each gadget typically ends with a *ret* (return) instruction, which allows the program's execution to continue at the next address on the stack, possibly controlled by an attacker. By carefully selecting and combining these gadgets, complex behaviors can be constructed, enabling control over systems without the need to inject new code [4].

B. Other Code-Reuse Attacks

This section describes other types of code-reuse attacks that an attacker could utilize.

1) Return-To-Libc: Return-to-libc is a technique used to bypass security measures by exploiting existing functions in the C standard library (libc). Instead of injecting new code, an attacker can manipulate the program's stack via a buffer overflow to overwrite the return address, redirecting execution to a libc function. By supplying their arguments to these function calls, leveraging the program's existing library code to bypass defenses. [5].

2) Jump-Oriented Programming: Jump-oriented programming (JOP) is another code-reuse attack similar to ROP, where an attacker aims to create a chain of gadgets to take control of the system. The difference is that JOP abandons all reliance on the stack and ret instructions, instead, their gadgets end with a *jmp* (jump) instruction. Knowing how to do this attackers can use jump instructions to manipulate the program's control flow, bypassing defenses designed to protect against ROP attacks [6].

3) Call-Oriented Programming: Call-oriented programming (COP) is similar to both ROP and JOP, as all involve chaining gadgets to control a program's execution. But while ROP uses *ret* instructions and JOP relies on *jmp* instructions, COP uses gadgets that end with indirect calls. Comparable to JOP, COP does not depend on the stack, but instead, it uses memory-indirect locations to guide the program's flow. This method helps attackers achieve their objectives while bypassing defenses specifically designed to counter ROP [7].

C. ROP-Chain Generation

There are various ways to leverage ROP for exploiting a program. While gadgets can be identified and assembled into chains manually, several tools are available that automate this process. These tools first identify a set of gadgets from the program's binary and then propose chains designed to achieve specific goals.

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.

1) Hard-Code Based Chaining Rules: The simplest approach to automation of ROP-chains relies on hard-coded regular expressions [8], [9]. Tools that fall into this category are e.g. *Ropper* [10] and *ROPGadget* [11]. ROP-chain generators in this category work in two stages: first, they search the target binary for gadgets; and second, attempt to chain gadgets together in a predefined way [9]. Relying on hard-coded rules and limitations on 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 [8].

2) Heuristics Based-ROP-chaining: 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 [12]. 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 [12]. ROP-chain generators belonging to this category include *angrop* [13] and *ropium* [14].

3) Exploratory approach with Satisfiability Modulo Theories (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 *Gadget Synthesis* [8][15]. The design of *Gadget Synthesis*, as described by Schloegel et. al [8], 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 defined 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 [8], the solver may require considerable time to identify valid gadget chains.

One of the earliest fully automated ROP-chain generators, *Q* [16], uses symbolic execution to assist in discovering gadgets within a target binary, and it relies on weakest precondition checks to verify whether a given instruction sequence can be used as a specific gadget type. This process is related to the principles of SMT solving, as *Q* employs logical constraints to ensure that each gadget meets the required postconditions for its intended function. However, *Q* primarily focuses on verifying and categorizing gadgets based on their functionalities, such as load, store and arithmetic operations, using an SMT solver. It then chains the categorized gadgets using traditional compiler algorithms.

4) Exploratory Approach with Partial-Order Planning: Partial-order planning is a technique utilizing AI to find a sequence of gadgets that can be used to build a ROP- chain. This type of approach is used in the tool *Gadget-Planner* [17]. It differs from total-order planning that 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 [18]. Partial-order planning's 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 is flexible. Because of that the state space became significantly smaller and faster to search. Partial-order planning starts at the goal state and searches backward for gadgets to fulfill its preconditions [9].

III. Problem definition

The performance, usability and efficiency of ROP generators can significantly differ based on the size and structure of the target binary. This project aims to compare and measure the differences between various ROP-chain generation tools to better understand their capabilities, limitations, and efficiency across various binary file sizes and functionalities. The goal is to highlight the differences between the different ROP generators and assess their qualities.

IV. Research questions

To guide the study, the following research questions are formulated which address both quantitative and qualitative aspects of the generators:

1) How many gadgets can each selected generator identify in binary files ranging in executable size from 1 kB to 8 MB?
2) What is the time required by each selected generator to identify a ROP-chain in binary files ranging from 1 kB to 8 MB in executable size?
3) Is there variability in number of gadgets and execution time to first chain between multiple runs of the same generator on identical binary files?
4) What architectures are supported by the different ROP-chain generators?
5) What types of code reuse attacks are supported by the different generators?

V. Methodology

The methodology for this study is designed to empirically compare several ROP-chain generators based on both quantitative and qualitative factors. The goal is to evaluate the performance and capabilities of the selected ROP-chain generators when generating chains from various binary files of different sizes. The criteria for selecting the ROP-chain generators and binary files used in the study will be outlined, alongside an explanation of the testing setup. The methodology will describe how the experiments were conducted and establish the methods for collecting and evaluating the results.

A. Selecting ROP-chain generators

The ROP-chain generators were selected to create a diverse and representative subset of the available tools while supporting x64 Linux binaries. The 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 was used to find open-source candidates. The categorization presented by Zhong et al. in [12] was used for the categorization in this project, due to it being quite recent and comprehensive. This categorization includes three categories: Hardcode-based 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, 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	Hardcoded	No
Ropper	Hardcoded	No
angrop	Heuristics	No
Gadget Synthesis	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 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.

A binary file consists of multiple sections each with a specific purpose. The executable portion of a binary is just one part of its overall structure, containing the actual machine code that can be run by the processor. For this reason, the share of executable code within each binary was considered important, as this can influence how much of the binary is involved in the 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. Qualitative evaluation

In this study, the evaluation of the qualitative factors of the ROP-chain generators is based on a static analysis of the code base and reviewing available documentation for each tool. The qualitative evaluation focused on identifying supported architectures and exploring additional code reuse attacks. The

TABLE II
SELECTED BINARIES

Binary	Version	Size (kB)	Executable section size (kB)	Share of executable	Category
/usr/bin/gdb	Gdb 15.0.50.20240403-0ubuntu1	11470	7045	61.42%	Debugger
/usr/python3.12	Python3 3.12.3-0ubuntu2	7832	2955	37.73%	Library
/lib/x86_64-linux-gnu/libc.so.6	Libc6 2.39-0ubuntu8.3	2076	1567	75.49%	Library
/usr/bin/bash	GNU bash, version 5.2.21(1)-release	1430	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	896	578	64.48%	Server utility
/usr/bin/wget	1.21.4-1ubuntu4.1	460	279	60.72%	Network
/bin/tar	Tar 1.35+dfsg-3build1	422	293	69.33%	File Management
/bin/netstat	Net-tools 2.10-0.1ubuntu4	155	77	49.47%	Network
/bin/ls	Coreutils 9.4-3ubuntu6	139	83	60.01%	File listing
/bin/touch	Coreutils 9.4-3ubuntu6	95	56	59.02%	File Management
/bin/ping	ping from iputils 20240117	88	43	48.73%	Network
/sbin/ifconfig	Net-tools 2.10-0.1ubuntu4	78	43	54.62%	Network
/bin/sleep	Coreutils 9.4-3ubuntu6	35	13	38.42%	General utilities
/bin/free	procps 2:4.0.4-4ubuntu3.2	27	7.7	28.57%	Memory Management
/bin/kill	procps 2:4.0.4-4ubuntu3.2	23	4.1	17.76%	Process Management
/usr/bin/clear	6.4+20240113-1ubuntu2	15	1.4	9.13%	Shell utility

documentation for the tools varies in source and comprehensiveness, where only a *README.md* file in the GitHub repository is obtainable for non-academic tools, but academic tools also have an associated research paper available. The analysis of the code base supplements the documentation review to find discrepancies and validate what is stated by the authors and the actual functionality the generator provides in the context of the qualitative factors.

D. Test environment

The test environment is built using the open source container service provider *Docker* where *Docker Compose* is utilized to manage multiple containers. Each container runs a service representing a ROP-chain generator, except for one dedicated container responsible for collecting the binaries. This binary container installs the binaries in specified versions within a base image and copies them to a shared directory mounted as a *Docker* volume. This shared volume enables all ROP-chain generator containers to access the binaries in a consistent version, as previously presented in Table II. The binary container uses a Dockerfile based on the *Ubuntu 24.04* operating system.

The purpose of this container setup is to establish a clean, isolated test environment where each generator can operate independently but under identical conditions, accessing the same set of binaries through the shared volume. By isolating each tool in its own container, the setup avoids conflicts in dependencies and configurations which can vary across ROP-chain generators.

The experiments were run on a Dell T5810 with an Intel Xeon E5-1650 v4 3.6GHz with 64GB DDR4 2400 MT/s running Proxmox Virtual Environment. The virtual machine used to run the experiments had the following resources allocated.

- CPU: Intel Xeon E5-1650 v4 10-threads 3.6GHz
- Memory: 60GB 2400MT/s DDR4 RAM
- Hard Drive: 1TB SDD
- Operation system: Ubuntu 22.04

E. Experiment setup

The primary objective of the experiment is to evaluate the ability of the selected ROP-chain generators to construct `execve` chains. These chains are a fundamental type of ROP-chain which is commonly used to execute a system call to spawn a shell on the target system. Focusing on `execve` chains ensures consistency in the evaluation across all tools, as this type of chain represents a standard use case in ROP-based exploits. The setup involves specifying the target function (`execve`) as the desired goal for each generator and assessing their ability to construct such a chain of gadgets.

Without any system calls in a binary, it is infeasible to construct `execve` chains. To address this issue, the executable `.init` section of each x86-64 compiled binary is injected with a system call gadget containing a *syscall* and *ret* instruction. The syscall instruction is essential for invoking system calls within the chain, and its availability significantly impacts the success rate of chain generation. By injecting such gadgets, the experiment eliminates variability in the availability of critical gadgets, allowing for a fair comparison of the capabilities of the tools. This also ensures that the focus remains on the ability of each generator to identify and chain gadgets rather than being limited by the specific properties of the binaries used.

F. Execution and data collection

To evaluate the performance and capabilities of the selected ROP-chain generators, each binary was executed 15 times on each generator. The time required to generate the first chain is extracted from the log files produced during each run. This data is used to address the research question concerning the time to generate the first chain. The logs are also analyzed to extract the total number of gadgets identified by each generator for a given binary. This metric supports answering the research question regarding the number of gadgets found across binaries of different sizes. To evaluate the variability in the number of gadgets and the time to generate a chain, the mean and standard deviation are calculated from the 15 runs for each binary-generator pair. This analysis provides insights into the consistency of the generators.

To ensure the feasibility of the quantitative evaluation, a preliminary test was conducted on each generator using the *ls* binary as a benchmark. The purpose of this test was to determine whether the generator could complete its execution within a one-hour time limit. Generators that failed to meet this time limit were excluded from the main experiment. Importantly, the one-hour time limit applied only to this initial screening process and was not enforced during the actual experiments for the generators that passed the preliminary test.

VI. Result

This section provides the results of the qualitative and quantitative evaluation of the different ROP-chain generators included in the study.

A. Quantitative comparison

One of the tools evaluated, *Gadget-Planner* [17], exceeded the one-hour time limit for the binary *ls* (see Table II), failing the preliminary test stated in V-F. Consequently, the generator was not included in the main experiment.

The experiments revealed that the generators were able to find chains of gadgets in only a subset of the 17 binaries tested, as summarized in Table III. Among the binaries, *python3* and *gdb* were the only ones for which all generators successfully produced chains. In contrast, the generator *ROP-gadget* failed to generate a chain for *libc* and *tar*, even though these binaries were successfully handled by the other tools. Notably, *ropium* demonstrated the highest compatibility, successfully generating chains for 7 binaries, more than any other generator included in the study, while *Gadget Synthesis* was only able to construct a chain in the *libc.so.6* binary.

Gadget Synthesis was the only tool in the experiments to trigger its built-in timeout. The generator timed out for binaries larger than *netstat*, except for *libc.so.6*. Additionally, it crashed for several binaries, such as *clear* and *tar*.

Further results for each generator will be presented in the following sections.

1) angrop: The binaries for which *angrop* was successful in generating chains, along with the mean time to the first chain and the standard deviation, are presented in Table IV. The binary *tar* exhibited the shortest mean time and the lowest standard deviation among all tested binaries, whereas *gdb* had the highest mean time and standard deviation, significantly surpassing the others.

The executable size of the binaries had a notable impact on the mean time, with larger executable sizes resulting in longer times to find the first chain, as illustrated in Figure 1. This trend was also observed for most binaries concerning the mean number of gadgets identified by *angrop*, where larger executable sizes generally resulted in more gadgets, as shown in Figure 2. However, notable discrepancies were observed with the binaries *ping*, *ifconfig* and *touch*. For instance, despite *ping* and *ifconfig* having the same executable size, *ifconfig* yielded a higher number of gadgets found. Additionally, *ifconfig* identified more gadgets than *touch*, even though *touch* has a larger executable size. Moreover, *openssl*, regardless of having an executable size larger than *ls*, *netstat*, *wget*, and *tar*, consistently identified fewer gadgets on average compared to these binaries.

As shown in Table V, the number of gadgets identified across tests exhibited slight variation for the binaries *tar*, *sshd*, *openssl*, *bash*, *libc*, *python3*, and *gdb*.

TABLE III
The binaries that respective generator found a chain in.

Binaries	angrop	ROPgadget	ropium	Ropper	Gadget Synth.
clear	✗	✗	✗	✗	✗
kill	✗	✗	✗	✗	✗
free	✗	✗	✗	✗	✗
sleep	✗	✗	✗	✗	✗
ifconfig	✗	✗	✗	✗	✗
ping	✗	✗	✗	✗	✗
touch	✗	✗	✗	✗	✗
ls	✗	✗	✗	✗	✗
netstat	✗	✗	✗	✗	✗
tar	✓	✗	✓	✓	✗
wget	✗	✗	✓	✗	✗
sshd	✗	✗	✓	✗	✗
openssl	✗	✗	✗	✗	✗
bash	✗	✗	✓	✓	✗
libc.so.6	✓	✗	✓	✓	✓
python3	✓	✓	✓	✓	✗
gdb	✓	✓	✓	✓	✗
	4 / 17	2 / 17	7 / 17	5 / 17	1 / 17

TABLE IV
angrop Time to First Chain Mean and Standard Deviation

Binary Name	Mean Time [s]	Standard Deviation [s]
tar	83.91	1.16
libc.so.6	573.3	1.6
python3	769.69	3.15
gdb	2290.14	44.23

TABLE V
angrop Number of Gadgets Mean and Standard Deviation

Binary Name	Mean Gadgets	Standard Deviation
tar	862.67	0.49
sshd	1264.73	0.46
openssl	675.47	0.64
bash	2777.2	0.86
libc.so.6	6224.47	0.64
python3	10176.8	1.37
gdb	16240.13	1.13

2) ROPgadget: The tool *ROPgadget* only successfully generated chains for the binaries *python3* and *gdb*. Among these, *python3* had roughly half the mean time to the first chain and nearly one-quarter of the standard deviation in time compared to *gdb*. Due to the limited number of binaries for which *ROPgadget* was able to generate chains, the relationship

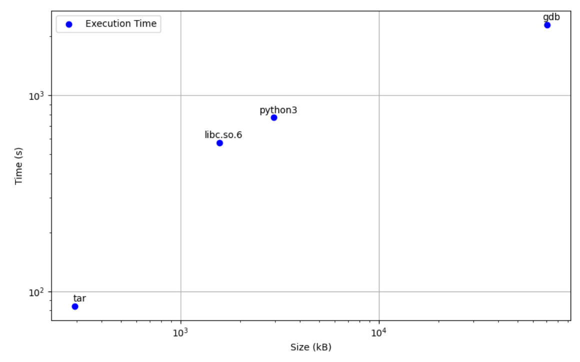

Fig. 1. Time to first chain for binaries successfully processed by *angrop*. Both axes are presented on a logarithmic scale.

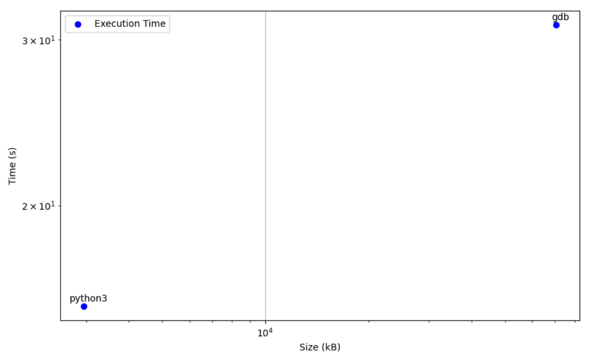

Fig. 3. Time to first chain for binaries successfully processed by *ROPgadget*. Both axes are presented on a logarithmic scale.

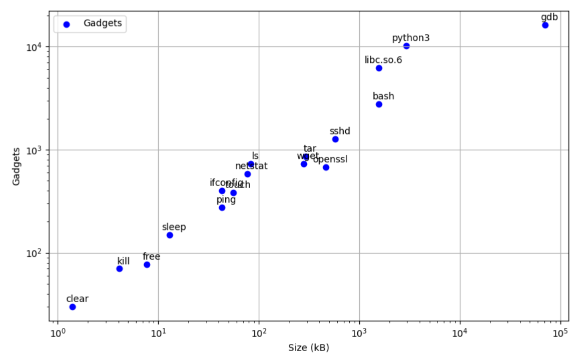

Fig. 2. Number of gadgets found by *angrop*. Both axes are presented on a logarithmic scale.

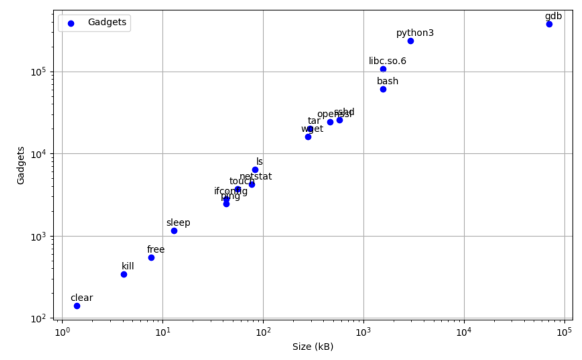

Fig. 4. Number of gadgets found by *ROPgadget*. Both axes are presented on a logarithmic scale.

between execution time and executable size, as presented in Figure 3, did not provide as much insight as observed with *angrop*.

TABLE VI
ROPgadget TIME TO FIRST CHAIN MEAN AND STANDARD DEVIATION

Binary Name	Mean Time [s]	Standard Deviation [s]
python3	15.63	0.71
gdb	31.09	3.56

Figure 4 illustrates the mean number of gadgets *ROPgadget* identified relative to binary size. The tool demonstrated a pattern similar to *angrop*, with the number of gadgets increasing according to a polynomial relationship as the executable size of the binary increased. Additionally, the number of gadgets found between runs indicated no variability.

3) ropium: The binaries for which *ropium* successfully generated a chain are presented in Table VII, along with the mean time to the first chain and the standard deviation between the 15 runs. The tool was successful in generating a chain for seven binaries, with the smallest binary being *tar* with an executable size of 422 kB. On the whole, *ropium* was able to generate chains for all binaries larger than *tar*, except for the *opnessl* binary. Likewise, the size of the binary also had an impact on the time for the first chain, as depicted in Figure 5 where it exhibited polynomial growth compared to the executable size of the binary. While a polynomial relationship can be observed in the number of gadgets the generator can find in each binary, as presented in Figure 6. A notable discrepancy to the growth was the jump between the *bash* and the *lib.c.so.6* binaries which are similar in size, but the tool can find more than twice as many gadgets in *lib.c.so.6* than in *bash*. Furthermore, there was no variance between each run for *ropium*.

4) Ropper: The tool *Ropper* successfully generated chains for five binaries: *tar*, *bash*, *libc*, *python3*, and *gdb*. As shown in Table VIII, *tar* exhibited the highest variation in execution time to the first chain, despite having the smallest executable size among these binaries. Furthermore, *libc* demonstrated a relatively high standard deviation in execution time compared to its size, in contrast to the other binaries except *tar*.

TABLE VII
ropium TIME TO FIRST CHAIN MEAN AND STANDARD DEVIATION

Binary Name	Mean Time	Standard Deviation [s]
tar	2.63	0.07
wget	2.2	0.05
sshd	3.03	0.05
bash	5.87	0.19
libc.so.6	11.86	0.21
python3	17.64	0.24
gdb	41.02	0.78

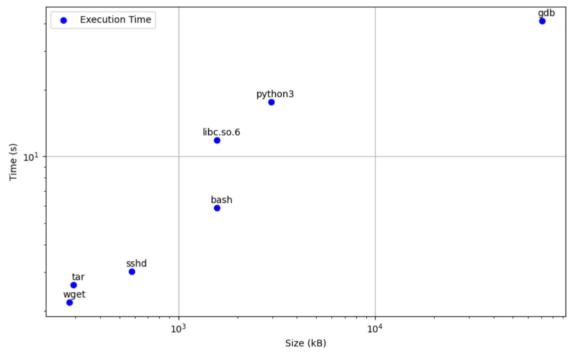

Fig. 5. Time to first chain for binaries successfully processed by *ropium*. Both axes are presented on a logarithmic scale.

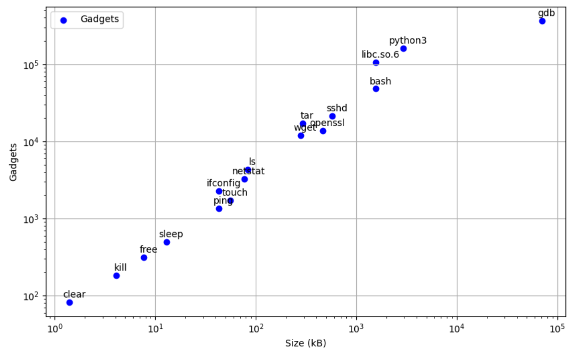

Fig. 6. Number of gadgets found by *ropium*. Both axes are presented on a logarithmic scale.

TABLE VIII
Ropper TIME TO FIRST CHAIN MEAN AND STANDARD DEVIATION

Binary Name	Mean Time [s]	Standard Deviation [s]
tar	3.88	1.43
bash	7.3	0.47
libc.so.6	14.74	1.24
python3	17.78	0.48
gdb	58.7	1.3

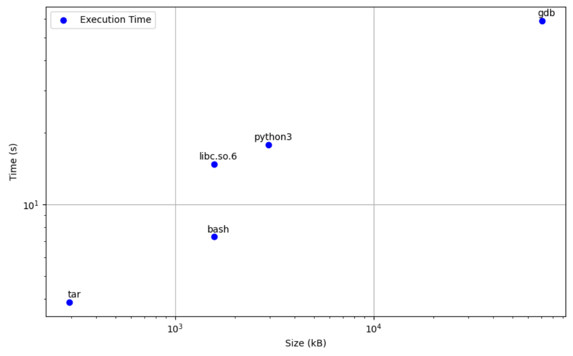

Fig. 7. Time to first chain for binaries successfully processed by *Ropper*. Both axes are presented on a logarithmic scale.

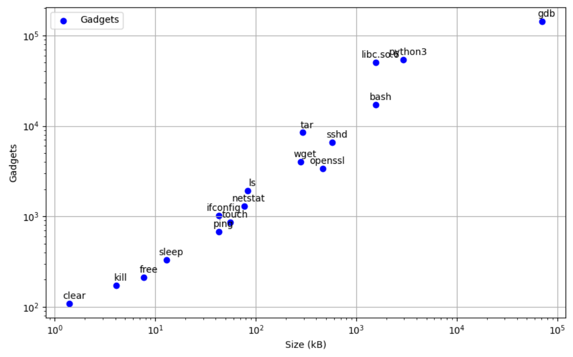

Fig. 8. Number of gadgets found by *Ropper*. Both axes are presented on a logarithmic scale.

The mean execution time to the first chain for *Ropper* follows a pattern similar to that observed in other generators, as illustrated in Figure 7. Notably, the mean execution time for the binary *libc* shows a significant increase compared to *bash*, which was also observed with the tool *ropium*, as shown in Figure 5.

The mean number of gadgets found by *Ropper* compared to the executable size in each binary is presented in Figure 8. The binary with the smallest executable size (*clear*) contained the least amount of gadgets and the binary with the largest executable size (*gdb*) contained the most. On average, *ifconfig* contained more gadgets than *touch*, despite having a smaller executable size. The same pattern can be seen for *tar*, *wget*, *openssl* and *sshd*.

There was no indication of any variability between runs concerning the number of gadgets.

5) Gadget Synthesis: The generator *Gadget Synthesis* was only successful in generating chains for the binary *libc.so.6*. For the other binaries, *Gadget Synthesis* either was unsuccessful in finding a chain or failed to complete one of the two phases, gadget discovery or chain assembly, due to the tool's

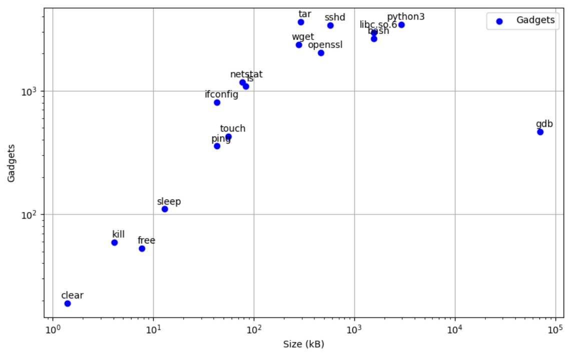

Fig. 9. Mean number of gadgets found by *Gadget Synthesis*. Both axes are presented on a logarithmic scale.

built-in one-hour timeout for each phase. For more information the reader is directed to Table XIII in the Appendix.

Table IX presents the mean of found gadgets in each binary, the variation between runs and how many of the runs for each binary completed. Additionally, the table indicates for which binaries the generator experienced a timeout in the gadget discovery phase or simply crashed during execution.

Figure 9 illustrates the mean number of gadgets *Gadget Synthesis* identified relative to the binary executable size. *Gadget Synthesis* demonstrated a similar pattern to the other tools, with the number of gadgets growing according to a polynomial with the executable size of the binary, but from binaries larger than *netstat* (size 159kB) the executable size does not seem to significantly affect the number of gadgets.

TABLE IX
Gadget Synthesis NUMBER OF MEAN GADGETS, STANDARD DEVIATION AND THE NUMBER OF TIMEOUTS TRIGGERED.
* = THE TOOL'S TIMEOUT WAS TRIGGERED IN THE GADGET DISCOVERY PHASE.
** = THE TOOL'S TIMEOUT WAS TRIGGERED IN THE GADGET DISCOVERY PHASE AND THE TOOL CRASHED IN THE CHAIN DISCOVERY PHASE.
*** = THE TOOL CRASHED IN THE CHAIN DISCOVERY PHASE.

Binary Name	Mean Gadgets	Standard Deviation	Completed
clear***	19	0.0	0 / 15
kill	59	0.0	15 / 15
free	53	0.0	15 / 15
sleep	110	0.0	15 / 15
ifconfig	811	0.0	15 / 15
ping	356	0.0	15 / 15
touch	428	0.0	15 / 15
ls	1084	0.0	15 / 15
netstat***	1176	0.0	0 / 15
tar**	3615.40	171.65	0 / 15
wget**	2374.53	465.9	0 / 15
sshd*	3393.93	206.27	0 / 15
openssl**	2034.07	319.24	0 / 15
bash*	2646.73	206.44	0 / 15
libc.so.6	2956.33	276.87	15 / 15
python3**	3428.27	303.55	0 / 15
gdb**	463.4	125.08	0 / 15

B. *Qualitative comparison*

This subsection will present the findings of the in-depth quantitative comparison of the selected ROP-chain generators.

1) Supported code reuse attacks: The evaluation of possible code reuse attacks with each generator indicates, as seen in Table X, that all support the generation of the ROP-chain. The code base of *angrop* [13] suggests that it is not built to support other code reuse attacks and no indication of this is observed in the tool documentation.

According to Zhang et. al their tool *Gadget-Planner* can generate chains that do not necessarily end with *ret* instructions, where it can also leverage gadgets that include direct, indirect and conditional jumps [9], which is a type of jump-oriented programming chain attack. This is also the case for *ROPGadget* [11], which incorporates a JOP search engine in the tool.

In the code base of *ropium* [14] and *Ropper* [10] it was observed that the tools, in addition to the *ret* instruction tools, are also able to categorize *JMP* gadgets as well. This indicates that the tool can perform JOP chain attacks.

The only generator that supports ROP, JOP and COP is *Gadget Synthesis* [15]. It is explained by Schloegel et. al in the paper *Towards automating code-reuse attacks using synthesized gadget chain* [8] that the tool utilizes all of the aforementioned code reuse attacks for chain generation.

TABLE X
THE TYPE OF CODE REUSE ATTACKS SUPPORTED BY THE CHAIN GENERATORS.

Gadget type	angrop	Gadget-Planner	ROPgadget	ropium	Ropper	Gadget Synthesis
COP	✗	✗	✗	✗	✗	✓
JOP	✗	✓	✓	✓	✓	✓
ROP	✓	✓	✓	✓	✓	✓

2) Supported CPU architectures: The results of the supported architectures are divided into two tables. The supported architectures found in the documentation of each tool are presented in Table XI, while the results of supported architectures observed during code analysis are found in Table XII.

Zhang et al. mention in their paper that *Gadget-Planner* uses the binary analysis framework *angr* for disassembly [9]. The framework [19] can decompile binary code from various architectures into an intermediate state, called `VEX`, on which *Gadget-Planner* will perform the search operation. Therefore, *Gadget-Planner* is able to create chains for all architectures that *angr* supports [9].

Schloegl et. al [8] does not mention in their paper, that *Gadget Synthesis* supports any architecture other than the AMD X86-64 processor, for which they conducted their experiments. They do mention that they use the binary disassembler *Binary Ninja*[1] which does support binaries for various architectures,

[1]Features of *Binary Ninja* is available at https://binary.ninja/features/.

including x86-64, ARM and MIPS. It is, however, unclear if the generator also supports these architectures.

The architectures for the other generators are presented in their respective GitHub repository's *README.md* file. It is observed that the documentation of *ropium* states that the tool is "soon to be extended with ARM" [14].

TABLE XI
SUPPORTED CPU ARCHITECTURES FOR EACH GENERATOR FOUND IN THE DOCUMENTATION.
■ = USES ANGR FOR DISASSEMBLY.
□ = USES CAPSTONE FRAMEWORK FOR DISASSEMBLY.

CPU architecture	angrop ■	Gadget-Planner ■	ROPgadget □	ropium □	Ropper □	Gadget Synthesis
ARM	✓	✓	✓	✗	✓	✗
ARV8	✗	✓	✗	✗	✗	✗
MIPS	✓	✓	✓	✗	✓	✗
PowerPC	✗	✓	✓	✗	✓	✗
RISC V	✗	✓	✓	✗	✗	✗
Sparc	✗	✓	✓	✗	✓	✗
x86-64	✓	✓	✓	✓	✓	✓

From the static code review, it was determined that *Gadget-Planner* [17] is able to load the architecture of the binary dynamically during runtime, which is done by using the *angr* framework [19]. For example, the registers of the current architecture, on which the binary file is compiled, are fetched with an API call to the *angr* framework.

Many of the ROP-chain generators have a hard-coded implementation for different architectures. In the code base of *ROPGadget* [11] it was observed that the architectures supported are hard coded into the binary loader. The architectures for *ropium* [14] are also implemented as hard-coded structures and *Ropper* [10] implements them as classes. All of these generators utilize *Capstone*[2] framework for binary disassembly; however, they do not support all the architectures that *Capstone* itself supports.

Gadget Synthesis [15] utilizes a similar approach to the generators that are leveraging the *Capstone* framework, implementing a hard-coded structure for each architecture it supports. In the current version of the code, it only has the implementation for x86-64.

The result of which architectures are supported by each generator according to the code analysis can be seen in Table XII.

[2]Supported architectures for the *Capstone* framework can be seen here: http://www.capstone-engine.org/arch.

TABLE XII
SUPPORTED CPU ARCHITECTURES FOR EACH GENERATOR FOUND IN CODE ANALYSIS.
■ = USES ANGR FOR DISASSEMBLY.
□ = USES CAPSTONE FRAMEWORK FOR DISASSEMBLY.

CPU architecture	angrop ■	Gadget-Planner ■	ROPgadget □	ropium □	Ropper □	Gadget Synthesis
ARM	✓	✓	✓	✓	✓	✗
ARV8	✗	✓	✗	✗	✗	✗
MIPS	✓	✓	✓	✗	✓	✗
PowerPC	✗	✓	✓	✗	✓	✗
RISC V	✗	✓	✓	✗	✗	✗
Sparc	✗	✓	✓	✗	✓	✗
x86-64	✓	✓	✓	✓	✓	✓

VII. DISCUSSION

The discussion section aims to interpret the results obtained from the study in the context of the research questions and objectives. Limitations observed from generators and the approach of the study are also discussed.

A. Gadgets Found

The first question in this study sought to determine how many gadgets each selected generator could identify in binary files in varying sizes, from the smallest at 15 kB (*clear*) to the largest at 12 MB (*gdb*). The result of the study confirms the association between larger binaries and the number of identified gadgets for a giving binary for all generators, see Figures 2, 4, 6 and 8. Likewise, the growth for all generators was polynomial for all binaries, which includes a possibility of following a linear trend. *ROPgadget* consistently finds the most gadgets across all binaries, whereas *angrop* is the tool that identifies the least amount of gadgets. For example for the largest binary, *gdb*, *ROPgadget* is able to identify 23 times as many gadgets as *angrop*. Surprisingly, *ropium*, which uses *ROPgadget* for the gadget discovery phase, identifies slightly fewer gadgets than *ROPgadget*. This could be due to the developers of *ropium* choosing slightly different parameters for running the tool than was done in this study. Even though *Ropper* and *angrop* use the same framework to disassemble binaries, the *angr* framework, *Ropper* was able to find a magnitude more gadgets than *angrop*. But still less than the tools that use the *Capstone* framework for binary disassembly. The discrepancy could also be explained by the different types of gadgets the tools support, as seen from Table X, *angrop* is only able to identify the ROP-type gadget, whereas *Ropper* also supports JOP-type gadgets.

Notably, the one-hour timeout during the gadgets disassembly process in *Gadget Synthesis* seems to impact the otherwise polynomial behavior observed in the rest of the generators. In Figure 9 the polynomial relationship can be seen for binaries not experiencing a timeout, while further increase in executable size does not significantly affect the number of gadgets found.

B. Time to First Chain

When addressing the second research question of this study, it is important to note a significant limitation: the generators successfully found a chain in only a small subset of the seventeen binaries tested. As a result, it is challenging to provide a comprehensive answer regarding their performance across the full range of executable file sizes, from 1 kB to 8 MB. However, meaningful insights can still be drawn from the observed results.

Notably, the generators *angrop*, *ROPgadget*, *ropium*, and *Ropper* demonstrated consistent behavior in execution time for finding the first chain. As illustrated in Figure 1, 3, 5 and 7, each presented on logarithmic axes, there is a polynomial relationship between the executable size of a binary and the execution time required to generate a chain. Larger binaries, such as *gdb*, consistently required the most time across all tools, whereas smaller binaries, such as *tar*, had the shortest execution times for generators like *angrop*, *ropium*, and *Ropper*.

The performance of *ROPgadget* was distinct, with the best results observed for *python3* (executable size: 2955 kB, mean time: 15.63 seconds) and *gdb* (executable size: 7045 kB, mean time: 31.09 seconds). Meanwhile, *ropium* slightly outperformed *Ropper* concerning finding a chain in binaries with executable sizes between 293 kB and 7045 kB, achieving mean times ranging from 2.63 to 41.02 seconds. In comparison, *Ropper* exhibited mean times ranging from 3.88 to 58.7 seconds within the same size range. The generator *angrop*, however, showed significantly poorer performance, with mean execution times ranging from 862.67 seconds to 16240.13 seconds for the same range of executable sizes.

As presented in the results, *Gadget Synthesis* was only able to assemble a chain for one of the 17 binaries tested, specifically *libc*. This outcome can be attributed to the unique properties of *libc*, which contains many of the essential UNIX programs and system calls needed for constructing chains [20]. Additionally, *Gadget Synthesis* mitigates performance issues by utilizing only a subset of the identified gadgets, 100 to 300, when processing binaries with a large number of gadgets [8]. While this approach helps reduce computational overhead, it may have contributed to the tool's inability to find chains in the other binaries.

C. Variability between Runs

This section examines whether the number of gadgets identified and the execution time to the first chain vary between multiple runs of the same generator on identical binary files.

1) Number of Gadgets: Among the five generators tested, two exhibited variability in the number of gadgets identified, *angrop* and *Gadget Synthesis*. For the latter generator, this variability was observed exclusively in binaries that timed out during the gadget discovery process, which were larger ones. In the case of *angrop*, deviations could be due to its approach to gadget identification. Additionally, when deviations occurred, they were often greater for larger binaries, although there were a few exceptions. System conditions, such as other running processes, may impact the performance of *Gadget Synthesis*, causing it to find fewer gadgets before a timeout and contributing to the observed variability. This behavior can be seen in Table IX.

2) Execution Time to First Chain: All generators showed some degree of variability in the execution time to generate the first chain, though not consistently across all binaries. Similar to the number of gadgets, the deviations in execution time were often more significant for larger files, with occasional exceptions. System conditions, such as other running processes, may have influenced performance even in cases where no timeouts occurred, contributing to this variability.

D. Supported Architectures

1) Architecture discrepancy: The results show an inconsistency regarding the supported architectures for *ropium* [14]. As presented in Table XI, *ropium* does not support the ARM architecture based on what was found in the documentation. However, the code analysis indicated that ARM, on the contrary, is supported. One reasonable argument for the discrepancy is that the code base may include partial support or groundwork for the ARM architecture, such as placeholder functions or initial implementation efforts. The analysis may have identified elements of ARM support without evaluating whether they are operational or integrated into the main functionality of the tool and the architecture support might not be fully implemented. This would explain why the documentation lists ARM as a future expansion of the tool.

Another possibility is a lack of synchronization between the code and the documentation. As highlighted by Emad Aghajani et. al [21], a delay in documentation updates after code development is a frequent issue in open-source projects which could apply to the *ropium* tool.

Supplying an answer to whether *ropium* supports the ARM architecture would require further analysis, for example, running experiments on the *ropium* generator with ARM-compiled binaries.

2) General observations: The results indicate a significant variation in the number and types of architectures supported by different generators. Tools leveraging the *angr* framework, such as *angrop* and *Gadget-Planner*, demonstrate broader architecture support, likely due to *angr*'s ability to abstract and handle multiple architectures dynamically. In contrast, tools like *Gadget Synthesis* and *ROPgadget* rely on hard-coded implementations, limiting their flexibility to specific architectures.

Most tools support common architectures like x86-64 and ARM, reflecting the dominance of these platforms in the current computing landscape. However, support for architectures such as RISC-V is limited, with only *Gadget-Planner* and *ROPgadget* showing support. This indicates potential areas for future development to accommodate the growing adoption of RISC-V [22] in both academic and industry contexts.

E. Type of Code Reuse Attacks

In the analysis, the types of code reuse attacks supported by the different generators were evaluated, specifically focusing on Call-Oriented Programming (COP), Jump-Oriented Programming (JOP), and Return-Oriented Programming (ROP). While ROP was universally supported across all tools tested, JOP was implemented in all but one generator, *angrop*. COP, on the other hand, was supported exclusively by *Gadget Synthesis*.

F. Limitations

It is important to be aware of the limitations of the approach this study employed to evaluate the qualitative and quantitative properties of the ROP-chain generators, as well as the limitations some generators have displayed.

1) Time Constraints and Tool Exclusion: The results demonstrated that *Gadget-Planner* [17] was unable to meet the time constraints set for the experiments. The tool exceeded the one-hour time limit when running the binary *ls* (see Table II). This led to the decision to exclude *Gadget-Planner* from further testing, as completing the experiments within the defined timeframe was not feasible.

A similar challenge was observed with *Gadget Synthesis*, which failed to process binaries with larger executable size than about 90 kB, except *libc*, within the set time limit. This limitation highlights the tool's inability to scale effectively with increasing binary sizes, which must be considered when evaluating its practicality for real-world applications involving large programs.

Although all generators demonstrated an polynomial increase in runtime as the executable binary size grew, the academic tools seem to have displayed a greater issue concerning the time to complete the execution and significantly limits their usability for larger binaries. These issues might stem from the increased complexity of finding gadgets and ways to assemble them into chains.

This experience underscores the importance of time efficiency in automated ROP-chain generation tools. Long runtimes not only hinder scalability but also become a bottleneck in environments where quick analysis is critical. Future work could explore optimization strategies for tools like *Gadget-Planner* and *Gadget Synthesis*, potentially improving their scalability and practicality for larger binaries. However, such optimizations were beyond the scope of this study.

2) Lack of qualitative testing: The approach of combining manual code analysis with documentation review was chosen to enhance the reliability of the results. However, there are some challenges using this approach. For instance, in manual code analysis, the subjectivity of the reviewer must be taken into account. Also, the documentation varied significantly in scope and detail. Non-academic tools only had a basic README file accessible on GitHub [23], whereas the academic tools came with detailed associated research papers about a specific tool. Despite the additional resources available for academic tools, much of the information provided was not relevant to the specific research questions regarding supported architectures and additional code reuse attacks. A significant limitation of this study is the absence of concrete experiments to confirm the results obtained from the code analysis and documentation review. Without validation, the findings may not fully capture the actual capabilities of the tools in real-world scenarios.

VIII. Conclusion

This study aimed to evaluate the capabilities of ROP-chain generators by answering the predetermined research questions.

- **How many gadgets can each selected generator identify in binary files ranging in executable size from 1 kB to 8 MB?**
  The study revealed that the number of gadgets identified by the selected ROP-chain generators generally increased with the size of the binary, confirming a polynomial pattern across all tools. Among the generators, *ROPgadget* consistently identified the highest number of gadgets across all binaries, outperforming the other tools significantly. For instance, in the largest binary (*gdb*, 12 MB), *ROPgadget* identified 23 times more gadgets than *angrop*, which consistently found the least number of gadgets.
- **What is the time required by each selected generator to identify a ROP-chain in binary files ranging from 1 kB to 8 MB in executable size?**
  The study revealed a polynomial relationship between the executable size of a binary and the time required by each generator to find the first chain. This relationship is consistent across all generators, with larger binaries generally requiring more time. *ROPgadget* demonstrated the fastest times for finding chains with binaries like *python3* (mean time: 15.63 seconds) and *gdb* (mean time: 31.09 seconds). *ropium* achieved mean times ranging from 2.63 seconds to 41.02 seconds for binaries sized between 293 kB and 7045 kB, slightly outperforming *Ropper*. *Ropper's* mean execution times ranged from 3.88 seconds to 58.7 seconds for binaries within the same size range as *ropium*. It performed better than *angrop* but was slightly less efficient than *ropium*. *angrop* consistently exhibited the longest execution times, with mean times ranging from 862.67 seconds to 16240.13 seconds for binaries in the 293 kB to 7045 kB range. *Gadget Synthesis* successfully assembled a chain for *libc* binary with a mean time of 4664 seconds.
- **Is there variability in number of gadgets and execution time to first chain between multiple runs of the same generator on identical binary files?**
  The study revealed limited variability in the number of gadgets identified but was more evident in the execution time to generate the first chain when running the same generator multiple times on identical binaries. Only *angrop* and *Gadget Synthesis* showed variability in gadget counts, primarily in larger binaries or those that timed out during the gadget discovery phase. In contrast, all tested generators showed some variability in execution time, particularly with larger binaries. Despite these deviations, the variability did not significantly impact the overall results, and the generators demonstrated relative consistency across runs.

- **What architectures are supported by the different ROP-chain generators?**
  The study revealed the following about the architectures supported by the evaluated generators. *angrop* supports x86-64, ARM, and MIPS via the *angr* framework. *Gadget-Planner* utilizes *angr* for disassembly, making it compatible with all architectures supported by *angr*, including x86-64, ARM, MIPS, and others. *ROPgadget* hardcoded support for x86-64, ARM, MIPS, and PowerPC through the *Capstone* framework, though it does not support all architectures that *Capstone* itself supports. *ropium* limited to x86-64 but includes experimental support for ARM, as mentioned in its documentation. *Ropper* supports x86-64, ARM, MIPS, PowerPC, and Sparc, leveraging the *Capstone* framework. *Gadget Synthesis* restricted to x86-64, with no observed support for other architectures despite using *Binary Ninja*, which theoretically supports a wider range of architectures.
- **What types of code reuse attacks are supported by the different generators?**
  All evaluated tools supported ROP, while all tools except *angrop* demonstrated additional support for JOP. *Gadget Synthesis* uniquely supported COP, making it the most versatile in this regard.

References

[1] T. Bletsch, “Code-reuse attacks: New frontiers and defenses,” Ph.D. dissertation, North Carolina State University, 2011, ISBN: 9781124752976.

[2] R. Yefeng, K. Sivapriya, and Z. Xukai, “Survey of return-oriented programming defense mechanisms,” *Security and Communication Networks*, vol. 9, n/a–n/a, Dec. 2015. DOI: 10.1002/sec.1406.

[3] L. Allouche, *Return-oriented programming (rop) chain*, Accessed: 2024-10-11, 2021. [Online]. Available: https://medium.com/@li_allouche/return-oriented-programming-rop-chain-358878d8bb02.

[4] H. Shacham, *The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)*, Accessed: 2024-10-11, 2007. [Online]. Available: https://hovav.net/ucsd/dist/geometry.pdf.

[5] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, “On the expressiveness of return-into-libc attacks,” in *RAID 2011, Lecture Notes in Computer Science, vol 6961*, R. Sommer, D. Balzarotti, and G. Maier, Eds., Springer-Verlag, 2011, pp. 121–141. DOI: 10.1007/978-3-642-23644-0_7.

[6] Z. Liang, T. Wei, S. Sethumadhavan, and R. Sekar, “Jop: Jump-oriented programming,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, ACM, 2011.

[7] N. Carlini and D. Wagner, “Rop is still dangerous: Breaking modern defenses,” in *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, 2014. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-carlini.pdf.

[8] 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: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*, Springer, 2021, pp. 218–239. DOI: 10.1007/978-3-030-88418-5_11.

[9] 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)*, 2023, pp. 313–326. DOI: 10.1109/DSN58367.2023.00039.

[10] S. Schirra, *Ropper*, Accessed: 2024-10-10, 2024. [Online]. Available: https://github.com/sashs/Ropper.

[11] J. Salwan, *Ropgadget*, Accessed: 2024-10-10, 2024. [Online]. Available: https://github.com/JonathanSalwan/ROPgadget.

[12] N. Zhong, Y. Chen, Y. Zou, *et al.*, “Tgrop: Top gun of return-oriented programming automation,” in *European Symposium on Research in Computer Security*, Springer, 2024, pp. 130–152.

[13] Y. Shoshitaishvili, R. Wang, A. Dutcher, C. Kruegel, and G. Vigna, *Angrop*, Accessed: 2024-10-10, 2024. [Online]. Available: https://github.com/angr/angrop.

[14] B. Milanov, *Ropium*, Accessed: 2024-10-10, 2022. [Online]. Available: https://github.com/Boyan-MILANOV/ropium.

[15] Chair for System Security, Ruhr University Bochum, *Gadget synthesis*, Accessed: 2024-10-10, 2021. [Online]. Available: https://github.com/RUB-SysSec/gadget_synthesis.

[16] 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. [Online]. Available: https://www.usenix.org/conference/usenix-security-11/q-exploit-hardening-made-easy.

[17] UNH SoftSec Group, *Gadget-planner*, Accessed: 2024-10-10, 2023. [Online]. Available: https://github.com/softsec-unh/Gadget-Planner.

[18] O. Sapena, E. Onaindía, and A. Torreño, “Combining heuristics to accelerate forward partial-order planning,” in *ICAPS 2014*, Cited by: 2, 2014, pp. 25–34. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84955290780&partnerID=40&md5=4ff43738090f81f9c3de605a2ae7f7c1.

[19] T. angr project contributors, *Angr documentation*, Acessed: 2024-11-06, 2024. [Online]. Available: https://docs.angr.io/en/latest/.

[20] A. Bansal and D. Mishra, “A practical analysis of rop attacks,” *arXiv preprint arXiv:2111.03537*, 2021.

[21] E. Aghajani, C. Nagy, O. L. Vega-Márquez, *et al.*, “Software documentation issues unveiled,” in *2019*

IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 1199–1210. DOI: 10.1109/ICSE.2019.00122.

[22] J. Saidova, “Risc-v architecture and its role in the near future,” *Journal of Advanced Scientific Research (ISSN: 0976-9595)*, vol. 5, no. 9, 2024.

[23] GitHub Inc, *Github*, Accessed: 2024-10-10, 2024. [Online]. Available: https://github.com.

Appendix

TABLE XIII
How many mean gadgets each generator found in each binary.
*=The timeout was triggered in the gadget discovery phase.
**=The timeout was triggered in the gadget discovery phase and the tool crashed in the chain discovery phase.
***=The tool crashed in the chain discovery phase.

Binaries	angrop	ROPgadget	ropium	Ropper	Gadget Synthesis
clear	30	141	82	109	19***
kill	70	342	184	172	59
free	77	547	316	213	53
sleep	150	1154	500	329	110
ifconfig	403	2811	2284	1024	811
ping	274	2462	1348	682	356
touch	381	3726	1724	860	428
ls	730	6375	4323	1910	1084
netstat	581	4229	3277	1306	1176***
tar	863	20018	17066	8438	3615.4**
wget	731	15932	12055	4020	2374.53**
sshd	1265	25663	21146	6620	3393.93*
openssl	675	24105	13757	3388	2034.07**
bash	2778	60904	47965	17031	2646.73*
libc.so.6	6224	106877	105611	50662	2956.33
python3	10179	233690	161710	53563	3428.27**
gdb	16240	376809	365352	142587	463.4**

TABLE XIV
The average execution time to find a chain in a binary.

Binaries	angrop	ROPgadget	ropium	Ropper	Gadget Synthesis
clear	✗	✗	✗	✗	✗
kill	✗	✗	✗	✗	✗
free	✗	✗	✗	✗	✗
sleep	✗	✗	✗	✗	✗
ifconfig	✗	✗	✗	✗	✗
ping	✗	✗	✗	✗	✗
touch	✗	✗	✗	✗	✗
ls	✗	✗	✗	✗	✗
netstat	✗	✗	✗	✗	✗
tar	83.01	✗	2.55	6.5	✗
wget	✗	✗	2.28	✗	✗
sshd	✗	✗	3.02	✗	✗
openssl	✗	✗	✗	✗	✗
bash	✗	✗	5.79	6.79	✗
libc.so.6	572.46	✗	11.85	14.87	4804.27
python3	765.3	14.88	17.41	18.41	✗
gdb	2256.19	30.47	40.51	58.13	✗
	4 / 17	2 / 17	7 / 17	5 / 17	1 / 17

TABLE XV
Gadget Synthesis time to first chain mean and standard deviation.

Binary name	Mean Time [s]	Standard Deviation [s]
libc.so.6	4664.90	1137.63

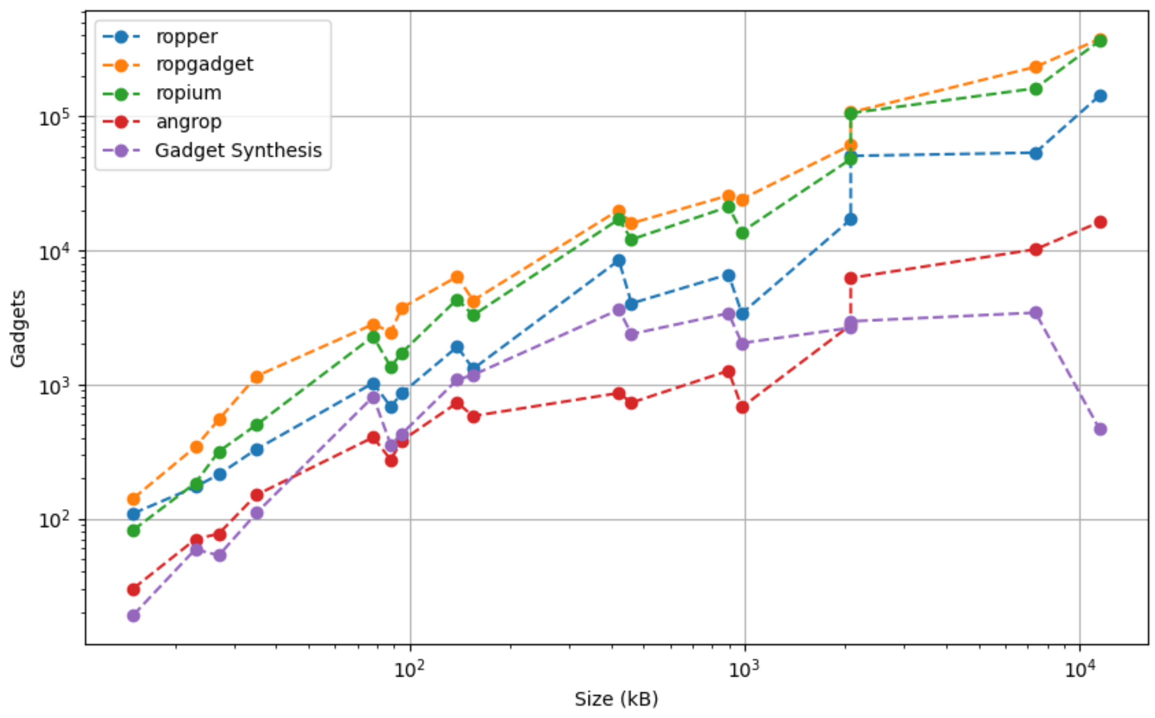

Fig. 10. The number of gadgets found by each tool in every binary.