USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

# Nothing is Unreachable: Automated Synthesis of Robust Code-Reuse Gadget Chains for Arbitrary Exploitation Primitives

Nicolas Bailluet, *Univ Rennes, Inria, CNRS, IRISA;* Emmanuel Fleury, *Univ Bordeaux, CNRS, LaBRI;* Isabelle Puaut and Erven Rohou, *Univ Rennes, Inria, CNRS, IRISA*

https://www.usenix.org/conference/usenixsecurity25/presentation/bailluet

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.


# USENIX Security '25 Artifact Appendix: "Nothing is Unreachable: Automated Synthesis of Robust Code-Reuse Gadget Chains for Arbitrary Exploitation Primitives"

Nicolas Bailluet
Univ Rennes, Inria, CNRS, IRISA

Emmanuel Fleury
Univ Bordeaux, CNRS, LaBRI

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA

Erven Rohou
Univ Rennes, Inria, CNRS, IRISA

## A Artifact Appendix

### A.1 Abstract

The paper addresses the problem of automatically chaining code-reuse gadgets to exploit vulnerabilities with arbitrary exploitation primitives. While related works generally require an attacker to control the stack (e.g. via a stack-overflow), our approach adapts to arbitrary situations – involving the stack, heap or anything else. Our experiments show that, unlike related gadget chaining approaches, our proof-of-concept tool – ARCANIST – supports various situations that involve diverse exploitation primitives and layouts – where competing tools are simply not usable. Especially, we demonstrate its applicability by generating gadget chains for 10 cases of real-world vulnerabilities involving non-trivial layouts. Additionally, we show that ARCANIST can generate long and complex chains, leveraging intricate chaining techniques – e.g. unassisted stack-pivoting, jump-oriented chaining and conditional instructions.

### A.2 Description & Requirements

The artifact is distributed as a set of archives. Among them, we provide `experiments.tar.gz`, which is the only one needed to reproduce the experiments. The rest of this document will exclusively refer to `experiments.tar.gz`. Other archives are provided for reference and transparency.

#### A.2.1 Security, privacy, and ethical concerns

No risk is taken when executing the artifact, no destructive steps are taken, and no security mechanism is disabled during execution. Moreover, experiments are containerized and are meant to be executed in the provided container to avoid polluting the evaluator's system.

#### A.2.2 How to access

The artifact is made available on Zenodo under the following DOI: 10.5281/zenodo.14724513.

#### A.2.3 Hardware dependencies

The tools run by the artifact may consume a lot of RAM and run on a lot of cores. If you don't have enough RAM, processes might get killed by the OS. If you don't have enough cores, this might affect execution times and overall performance. We recommend running the experiments on a system with at least **96GB of RAM**, and **ideally 64 CPU cores or more** (the specifications of our experimental setup). As a reference, *our CPU cores were running at 2.3GHz.*

#### A.2.4 Software dependencies

The artifact is meant to be run on a **Linux operating system**. The experiments are containerized to facilitate their execution, isolation, and dependencies installation. You will need **`Docker`** to run the containers and launch the experiments.

#### A.2.5 Benchmarks

The binaries used for our experiments are provided in the archive in two folders:

- `./cve_binaries` for the experiments of Section 8.1 to 8.3 on several CVEs.
- `./binaries` for the experiments of Section 8.4 to 8.6.

Our tool ARCANIST uses a commercial (non-free) version of Binary Ninja to lift gadgets. Therefore, to ensure reproducibility, and for other tools as well (when possible), we provide the gadget libraries already extracted – for ARCANIST in `./arcanist/gadgets`, for SGC in `./sgc/target/*/.cache`, and for Angrop in `./angrop/gadgets`.

## A.3 Set-up

### A.3.1 Installation

1. Install `Docker` if not already done and ensure that you can run `docker compose`.
2. Download and extract the `experiments.tar.gz` archive from the Zenodo repository (see A.2.2).
3. Inside the extracted folder, run the following command to import the containers:

```
$ ./import-containers.sh
```

### A.3.2 Basic Test

Test environments are containerized and ready-to-use, just check that you can access the containers by running:

```
$ docker compose run <TOOL>
```

with `<TOOL>` being `angrop`, `ropium`, `sgc` or `arcanist`.

In each container, a basic functionality test can be run with:

```
$ ./functionality_test.sh
```

In the end, it should print whether the basic functionality test passed or failed.

## A.4 Evaluation workflow

Note that, since ARCANIST launches multiple instances in parallel, the generated chains may sometimes be different from the ones given in the paper (an instance can succeed before another, and randomness is involved in the underlying solver).

### A.4.1 Major Claims

**(C1):** ARCANIST can generate gadget chains for situations where existing works are just unusable. This is proven by the experiment (E1) detailed in Section 8.1 of the paper, whose results are given in Table 1.

**(C2):** ARCANIST can generate long chains and leverage complex chaining techniques without explicit requests – such as jump-oriented chaining, unassisted stack-pivoting, use of conditional instructions and use of additional buffers. This is proven by the experiments (E2) and (E3), described in Sections 8.2 and 8.3 of the paper, whose results are illustrated in Listing 2, 3 and 4.

**(C3):** Even in situations where existing works are usable, ARCANIST proves to be more flexible and versatile by being able to reach more attack goals and succeeding more often than its competitors. This is proven by the experiment (E4), detailed in Section 8.4 of the paper, whose results are given in Table 2.

### A.4.2 Experiments

Items (E1) to (E4) are the most important ones, dedicated to our major claims. Regarding items (E5) to (E7), they are dedicated to – less significant – time-consuming performance evaluations.

**(E1):** *[CVEs (Table 1)] [5 human-minutes + 2 compute-hours]:* In this experiment, we ask ARCANIST to generate gadget chains for real-world CVE situations where other tools are unusable. We expect ARCANIST to succeed in generating chains in almost all situations – except one, see Table 1 in the paper.

**Execution:** Firstly, enter the `arcanist` container:

```
$ docker compose run arcanist
```

Then, run the following command to start the experiment:

```
$ ./run_table1_experiments.sh
```

**Results:** Once done, you can print the results with:

```
python print_table1_results.py
```

This will display the results in the same order as in Table 1 in the paper.

**(E2):** *[Complex Chains (Listing 2 & 3)] [5 human-minutes + 30 compute-minutes]:* In this experiment, we show the complexity of ARCANIST-generated chains through two examples. We expect ARCANIST to generate non-trivial chains showing: jump-oriented chaining, use of conditional instructions and stack-pivoting. The generated chains should be similar to the paper's chains, but you might observe variations.

**Execution:** Firstly, enter the `arcanist` container, then, run the following commands separately:

```
$ ./run_listing2_synthesis.sh
$ ./run_listing3_synthesis.sh
```

**Results:** For each command, the generated chain should be printed in the end, once done.

**(E3):** *[OPTEE Chain (Listing 4)] [5 human-minutes + 5 compute-minutes]:* In this experiment, we show the ability of ARCANIST to generate a stack-pivoting chains for our OPTEE CVE case-study. The generated chain should be similar to the paper's chain, but you might observe variations.

**Execution:** Firstly, enter the `arcanist` container, then, run the following command:

```
$ ./run_listing4_synthesis.sh
```

**Results:** Once done, the generated chain should be printed in the end.

**(E4):** *[Tools Comparison] [10 human-minutes + 15 compute-hours + 90GB-disk]:* This experiment compares ARCANIST with other tools in situations where its competitors are usable. We expect ARCANIST to always succeed, while other tools should fail in some situations.

**Execution:** In each tool's container, run the following command:

```
$ ./run_table2_experiments.sh
```

**Results:** The results for each tool can be printed by running the following command in the tool's container:

```
$ python print_table2_results.py
```

This will print the results in the same layout as the tool's associated column from Table 2 in the paper.

**(E5):** *[Memory Writes Benchmark] [10 human-minutes + 40 compute-hours]:* This experiment evaluates the performance of ARCANIST regarding the `mem-write-ratio` parameter. We expect to observe a gap in average execution times between ratios of 0% and 5% (most important observation). The average execution time is also expected to increase as the ratio increases from 5% to 25% – the paper shows a linear evolution, but depending on the experimental setup, the linear evolution might not be as clean as Figure 7 in the paper.
**Execution:** Enter the `arcanist` container, then, run the following command:

```
$ ./run_figure7_experiments.sh
```

**Results:** You can plot the figure by running:

```
$ ./plot_figure7.sh
```

This will generate a `pdf` at `./arcanist/synthesis_results/memory_writes_benchmark/avg_time_mem_writes.pdf`, that you can open outside the container.

**(E6):** *[Sample Size vs Jobs Benchmark (libc)] [10 human-minutes + 5 compute-days]:* This experiment evaluates the performance of ARCANIST regarding the `jobs` and `size` parameters, for a stack-based scenario in libc. We expect the best execution times to be achieved with samples of 100 gadgets (most important observation). We also expect to see a color gradient similar to the one given in Figure 8 in the paper.
**Execution:** Enter the `arcanist` container, then, run the following command:

```
$ ./run_figure8_experiments.sh
```

**Results:** You can plot the figure by running:

```
$ ./plot_figure8.sh
```

This will generate three `pdf` in `./arcanist/synthesis_results/lib_size_vs_process_count_evaluation/libc.so.6/`, named `average_time.pdf`, `success_ratio.pdf` and `average_chain_length.pdf`, which correspond to the three heatmaps in Figure 8. You can open them outside the container.

**(E7):** *[Sample Size vs Jobs Benchmark (dnsmasq)] [10 human-minutes + 7 compute-days]:* This experiment evaluates the performance of ARCANIST regarding the `jobs` and `size` parameters, for a heap-based scenario in dnsmasq. We expect the best execution times to be achieved with larger samples than (E6), and execution times should be one order of magnitude higher than (E6) (most important observation). We also expect to see a color gradient similar to the one given in Figure 9 in the paper, especially there should be a lot of black squares (indicating failures/timeouts).
**Execution:** Enter the `arcanist` container, then, run the following command:

```
$ ./run_figure9_experiments.sh
```

**Results:** You can plot the figure by running:

```
$ ./plot_figure9.sh
```

This will generate three `pdf` in `./arcanist/synthesis_results/lib_size_vs_process_count_evaluation/dnsmasq/`, named `average_time.pdf`, `success_ratio.pdf` and `average_chain_length.pdf`, which correspond to the heatmaps in Figure 9. You can open them outside the container.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.