USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Synthesis of Code-Reuse Attacks from p-code Programs

Mark DenHoed and Tom Melham, *University of Oxford*

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

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: Synthesis of Code-Reuse Attacks from `p-code` Programs

Mark DenHoed
University of Oxford

Tom Melham
University of Oxford

A Artifact Appendix

A.1 Abstract

Our artifact is composed of four components:

- `jingle`: a Rust library for modeling the semantics of a fragment of `p-code`, the Intermediate Language of the Ghidra decompiler, in the language of Satisfiability Modulo Theories (SMT).
- `crackers`: a Rust library for synthesizing Code Reuse Attacks (aka. ROP Chains) from a simple `p-code` program using models produced by `jingle` and a conflict-driven search powered by a Boolean Satisfiability (SAT) solver.
- Our evaluation setup comparing the performance of `crackers` to several open-source ROP synthesis tools. We include our raw data, utilities to generate our charts and tables, and the code needed to re-run the entire evaluation.
- A case-study demonstrating the use of `crackers` in synthesizing a ROP chain for CVE-2017-14493: a simple Remote Code Execution vulnerability in `dnsmasq`.

We include standalone copies of `jingle` and `crackers` for the record. While evaluators may install and use these tools, bundled copies of these libraries are used internally by our ROP tool evaluation and case-study. The standalone copies are not necessary to evaluate the claims of the paper. We will indicate instructions for the optional usage of these tools with the heading **`jingle` and `crackers` (optional)**.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifacts pose no risk to evaluator's machines.

A.2.2 How to access

Our artifacts can be accessed via Zenodo using the DOI 10.5281/zenodo.14738160

A.2.3 Hardware dependencies

While our real-world exploitation case-study is dockerized, it requires an ARM CPU due to its usage of GDB and `ptrace`. We tested it on an Apple M1 processor.

We don't require any specific CPU for `jingle`, `crackers`, or our ROP tool evaluation. As many of the evaluated tools use SMT Solvers, we recommend that our ROP tool evaluation be run on a system with at least 32 Gigabytes of RAM. We also recommend that at least 15 gigabytes of disk space reserved for Docker due to the large number of built containers.

A.2.4 Software dependencies

The ROP tool evaluation and the `dnsmasq` case study both run fully inside Docker and make use of Docker Compose, so Docker must be installed. We additionally recommend the Just command runner to simplify the usage of our Docker Compose setup, but this is not mandatory.

`jingle` and `crackers` (optional). Both tools require that the system have both the Ghidra and Z3 to be installed to run. If using a macOS system, the version of Z3 in Homebrew will work; if using a Debian-based linux, we recommend using the build distributed by the Z3 project, as Debian distributes a very outdated version of Z3. Additionally, both these libraries require the Rust compiler to be installed as well as your platform's standard C/C++ compiler/linker. We have used both tools on both x86 and ARM, as well as linux and macOS. The tools should work on Windows systems as well, but we have not verified this outside of a CI pipeline.

A.2.5 Benchmarks

Our ROP tool evaluation uses the ALLSTAR data-set as a source of test binaries. We do not include this dataset in the artifacts due to its size. Instead, our evaluation code retrieves binaries from JHU/APL's hosted version of the data-set as needed. These files are saved temporarily within a docker container and deleted afterwards to minimize the needed disk space. An active internet connection is therefore needed throughout the evaluation.

A.3 Set-up

A.3.1 Installation

ROP Tool Evaluation. In the `crackers_evaluation` folder, verify docker is running with `docker --version` and attempt to build all the images used in the evaluation with `docker compose build`. This will build a container for every ROP tool and algorithm ablation evaluated in the paper. The docker build should succeed with no errors.

`dnsmasq` Case Study (ARM ONLY). In the `dnsmasq_poc/arm64` folder, run `docker compose --profile poc build`. This should build both a container with a vulnerable version of `dnsmasq` and a container named `poc` containing a tool for exploiting this vulnerablity using `crackers`. The docker build should succeed with no errors.

`jingle` and `crackers` (optional). Install Rust, Z3, and Ghidra from the links provided above. Find the root directory for each tool (the highest one with a `Cargo.toml` file inside it). Now, for `jingle`, run `cargo install --path ./jingle --features="bin_features"`. For `crackers`, run `cargo install --path . --features="bin"`. There should now be two new binaries in your path, named `jingle` and `crackers`.

A.3.2 Basic Test

The basic ROP tool evaluation setup can be tested first by executing `just restore` or by manually running the following commands:

```
docker compose down redis
docker run -d --rm --name temp \
       -v crackers_evaluation_cache:/data \
       ubuntu sleep infinity
docker cp evaluation_data.rdb \
       temp:/data/dump.rdb
docker stop temp
docker compose up redis -d
```

This loads our original evaluation data into the testing infrastructure. Our original graphs can be produced by running:

```
docker compose run --rm grapher \
       all-tool.svg combined all-tool
docker compose run --rm grapher \
       ablation.svg combined ablation
```

This should produce two SVG files containing the charts from the paper with our results from (E1) and (E2). Clear the database of our evaluation data by running `just cleanup_db` or by manually running `docker compose exec redis redis-cli flushall`.

To verify that the `dnsmasq` case-study is ready, run `docker compose up dnsmasq` to verify that `dnsmasq` was built and runs.

`jingle` and `crackers` (optional). `jingle` can be easily tested with the following command: `jingle disassemble <path_to_ghidra> x86:LE:64:default 55`. This should produce the output `PUSH RBP`. Replace `disassemble` with `lift` to see the `p-code` associated with this x86 instruction, or with `model` to see `jingle`'s SMT modeling of this `p-code`. To test `crackers`, you can use a sample that is provided in the `crackers` directory. First, edit `crackers/sample_config.toml` and ensure that the `ghidra_path` parameter points to your Ghidra installation. You can then run `crackers synth sample_config.toml` and verify that `crackers` finds a ROP chain.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): `crackers` matches or exceeds the performance of the evaluated state-of-the-art ROP chain synthesis tools across a wide corpus of binaries. This is proven by the experiment (E1) described in Section 5.1 whose results are illustrated in Figure 3 and Table 1.

(C2): `crackers` matches or exceeds the performance of the evaluated algorithm ablations across a subset of our corpus of binaries. This is proven by the experiment (E2) described in Section 5.2 whose results are illustrated in Figure 4 and Table 2.

(C3): `crackers` is capable of synthesizing a working ROP chain within the constraints of a real-world vulnerability. This is proven by the experiment (E3) described in Section 5.3.

A.4.2 Experiments

(E1): ROP Tool Comparison [1 human hour + 2 compute-weeks + 30GB Disk + 32GB RAM]: A comparison between `crackers` and several ROP chain synthesis tools over a large corpus of binaries demonstrating `crackers`' equivalent or superior run-time performance and success-rate.

How to: This evaluation is performed through our docker-based testing infrastructure. Each ROP tool is evaluated in its own docker container with results automatically reported and organized in a shared `redis` database. We recommend executing the tools serially to avoid multiple tools' memory and CPU usage confounding timing measurements. We also provide dockerized tools to export the contents of the database into charts and tables of the kind used in our paper, allowing for easy comparison with our results.

Preparation: No additional configuration is required beyond the items already covered in previous sections.
Execution: This evaluation compares `crackers` to three other ROP tools: `angrop`, `exrop`, and SGC. To run this evaluation, you will run one docker command per tool in series (the commands themselves are explained in more detail in a README in our artifacts). In our evaluation, runtimes of each command ranged from around 12 hours to 10 days.
Results: Once the evaluations have finished, you can run our dockerized charting and statistics utilities to produce a version of Figure 3 and Table 1 derived from your new data. These reporting commands can also be run from another terminal at any time during the experiment. This allows for monitoring of progress and truncated runs. The commands to use these dockerized utilities are explained in more detail in the README.
Due to known malformed data in the ALLSTAR dataset, the evaluation runner will periodically fail to decode a package, indicated by the message "`error decoding response body`". This is expected and does not affect the results of the evaluation.
The test runner may occasionally fail to fetch an ALLSTAR package due to networking issues. We include a command to identify binaries in the `crackers` dataset that are missing from that of another ROP tool. Run `docker compose run --rm stats missing <tool>` to receive a summary of missing binaries for the given tool, as well as docker commands to re-attempt evaluation of those specific test cases.

(E2): Ablation Study [1 human hour + 2 compute-weeks + 15GB Disk + 32GB RAM]: This experiment evaluates `crackers` relative to two algorithmic ablations: `crackers_a` and `crackers_b`.
How to: This experiment uses the same Docker Compose setup as (E1), run over a separate set of Docker containers, each containing an ablation of `crackers`.
Preparation: As in (E1), no additional configuration is required beyond the items already covered in previous sections.
Execution: Like (E1), this experiment is conducted by executing several long-running docker commands. In our evaluation, runtimes of each command ranged from around 12 hours to 10 days.
Results: These results are validated using the same method as in (E1). The chart and table produced by this experiment can be compared to Figure 4 and Table 2 respectively.

(E3): Exploitation Case Study [15 human-minutes + 15GB Disk + 15 compute-minutes (ARM ONLY)]: This experiment demonstrates the use of `crackers` in an exploitation tool targeting `dnsmasq` (CVE-2017-14493). We configure `crackers` to synthesize a ROP chain that cleanly exits the program with a controlled exit code, without throwing any signals.
How to: This evaluation is also performed using Docker Compose-based infrastructure, located in `dnsmasq_poc/arm64`. A vulnerable `dnsmasq` service is started in a container, listening for DHCP6 requests on a non-standard port. Our exploitation tool is run in a separate docker container, targeting the private IPV6 docker network address of the `dnsmasq` container. The container with the exploitation tool uses `crackers` to synthesize a ROP chain, which it uses to construct a DHCP6 packet that it sends to `dnsmasq`. If all works correctly, `dnsmasq` exits with our chosen error code.
Preparation: As in (E1) and (E2), no additional configuration is required beyond the items already covered in previous sections.
Execution: This is most easily executed with the Just command runner. First, start the `dnsmasq` container and connect its `dnsmasq` service to GDB by running `just debug`. Once gdb is attached, run `just poc` in another shell. This will build the exploit tool (if you have not already built the container), and begin synthesis of a ROP chain for the `dnsmasq` binary.
Results: When synthesis completes successfully, it will fire off the DHCP6 packet. You should observe `dnsmasq` exit cleanly without tripping GDB.

A.5 Notes on Reusability

Both `jingle` and `crackers` were written to enable further usage. Both tools are standalone Rust crates, exposing programmatic APIs as well as simple command-line tools.

Our Rust API for ALLSTAR is also written as a standalone Rust crate, and so can easily be used or extended in other applications.

Our ROP tool evaluation setup makes very few assumptions about the tools it runs and could easily be extended with additional ROP tools. It could also be easily adapted to measure run-time performance of any program analysis tool over portions of the ALLSTAR dataset.

A.6 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/.