USENIX Security '26 Artifact Appendix: Discovery, characterizing and exploiting controllable-copy objects for kernel data-only attacks with CopyKat

Jakob Koschel[†*] Andrea Mambretti[†*] Alessandro Sorniotti[†*] Pietro Moretto[†]
Claudio Migliorelli[†] Andrea Di Dio[‡] Cristiano Giuffrida[‡] Anil Kurmus[†]

[†]IBM Research Europe – Zurich [‡]Vrije Universiteit Amsterdam

[*]These authors contributed equally

A Artifact Appendix

A.1 Abstract

In this work, we build a tool, CopyKat, capable of finding, verifying and characterizing Controllable Copy Objects (CCOs) within the Linux kernel. These objects can be used to amplify exploitation primitives into more powerful ones. For instance, they can allow a limited out-of-bound write on the heap to turn into arbitrary write. In our work, we provide examples of such use through the construction of 8 end-to-end exploits. Having tools like CopyKat can help discovering and categorizing such sensitive objects and can help with constructing principled defenses.

CopyKat is composed of three different stages. The first component is a modified Syzkaller capable of identifying candidate reproducers that follow the *CCO pattern* introduced in our work. This component runs a lightweight taint analysis on top of an instrumented kernel and a modified version of KASAN. The second component performs a more precise taint analysis using PANDA, where we verify the correctness of the taint and remove false positives. The third component takes the verified reproducers, the more precise taint information, and runs the reproducer under concolic execution. This component symbolize three main areas of interest, and perform the simulated corruption of the CCO object. It verifies that the primitive highlighted by the taint is still reachable after corruption, and it provides the path-constraints that an attacker needs to satisfy to use such object as described by that specific reproducer.

For this artifact evaluation, we provide the instructions on how to use and reproduce our taint verification and concolic execution analysis and their results. While we provide the code patches (on Zenodo [1]) to run our fuzzing campaign through Syzkaller, we do not include it in this artifact evaluation. This choice is dictated by the fact that it would not be possible to guarantee that similar results would be found to compare against our paper findings, owing to the stochastic nature of fuzzing. Furthermore, such stage alone would require much more time than allowed to replicate our results. We provide the results of our fuzzing campaign as starting seeds to the two remaining analyses.

[1] https://doi.org/10.5281/zenodo.18500424

A.2 Description & Requirements

Our artifact contains two separate projects. Both are based upon large open-source projects like PANDA and S2E that in turns have several dependencies. To ease deployment, we use Docker to guarantee that such dependencies are easily satisfied.

While our paper proposes a pipeline, we have setup the two projects under evaluation as independent components to ease their inspection. Our taint analysis takes as input the reproducers from the fuzzing campaing and produces as output the object database. Meanwhile, the concolic execution takes as input the already generated PANDA database that we used during our paper evaluation and provides as output the path constraints of the verified objects.

Hereafter, we will indicate the requirements and the instructions for each stage of the pipeline separately.

A.2.1 Security, privacy, and ethical concerns

Our two analyses do not pose any security risk to the evaluator's machine beyond occupying large amount of disk space. The taint verification component requires to run `python` scripts and `gcc` on the evaluator's machine to transform and integrate the database throughout the experiment, and generate the executable reproducers from the `.cprog` files produced by Syzkaller. The PANDA analysis is run within a docker

container with no access to user data beyond our reproducers. Similarly, the concolic execution is also fully packaged within a docker container and none of its components is run directly on the evaluator's machine besides the `docker build` command.

A.2.2 How to access

Our main artifact to reproduce the paper results can be found on public Github at https://github.com/IBM/CopyKat. The repository hosts the code of our two CopyKat analyses and all necessary input data to re-run and obtain the results of our paper. Meanwhile, on Zenodo at https://doi.org/10.5281/zenodo.18500424, we provide all the code under patch format, including also the code for fuzzing campaign, and our own evaluation results the paper is based on.

A.2.3 Hardware dependencies

CopyKat is a tool developed for x86 architecture (Intel/AMD). It produces a large quantity of data and, if scaled to several CPU, it can be memory hungry. While the system would run on a small scale machine (min 8 cores and 16GB of RAM), for performance reasons, we recommend at least 16 cores and 32GB of RAM. The only true constraint is the disk space for which we suggest at least 200GB of free space to prevent issues.

A.2.4 Software dependencies

CopyKat was developed using Linux Ubuntu 22.04 LTS. We packaged CopyKat with Docker to reduce the burden of satisfying large set of dependencies. Our main software dependencies on the evaluator's machine are the latest `python3`, `docker`, `docker-compose`, `git`, `gcc` and `git-lfs` that can be downloaded on Ubuntu 22.04 LTS. Newer versions of Ubuntu could potentially work, but we did not test them.

A.2.5 Benchmarks

None

A.3 Set-up

As initial step, it is necessary to check if all the software dependencies are installed.

```
sudo apt get install python3 docker.io docker-compose
↪ docker-compose-plugin git git-lfs
```

Then, before cloning our repository, make sure to configure `git-lfs` on the system. Our repository contains few large files that cannot be directly stored on the Github repository.

```
git lfs install
git clone https://github.com/IBM/CopyKat.git
```

Unfortunately `git-lfs` has a monthly limit for download through `git clone`. If such limit is reached, we encourage to download the following files through the web interface.

```
# taint verification
linux/vmlinux

# concolic execution
s2e-data/database/panda.data
s2e-data/kernel/vmlinux
s2e-data/s2e-image/rootfs.ext2
s2e-data/s2e-image/rootfs.ext4.ready
```

A.3.1 Installation

We consider the installation of the two components separately. The two components can be found under the folders `taint_verification` and `concolic_execution_with_corruption` respectively.

Taint verification To successfully run our PANDA analysis, these steps must be followed. First, we need to use *static analysis* to extract some information from the Syzkaller reproducers by running:

```
./01.extract-info.py  --crashes ./repros/ --outfile
↪ db/01.data
```

Then, it is required to compile the reproducers into binary by invoking:

```
./02.compile.py --reports-file db/01.data --path repros/
↪ --outdir ./repros/
```

Now, we need to create the docker container for PANDA and connect to it.

```
cd panda/
docker compose up -d
docker exec -it panda-syz-rrr-1 bash
```

Our PANDA analyses are based on an execution recording of each reproducer. To generate these recordings that will be used to replay later, it is necessary to run:

```
./kdo/run-repros.py --reports-file db/01.data --path
↪ /root/repro/ --outfile db/record.1.json
```

This command will take a considerable amount of time. It needs to boot the QEMU snapshot and then sequentially record executions. The program generates a recording file called `db/record.1.json`. Recordings might fail: grepping the string "err" will reveal recording errors. This does not reflect any fundamental issues with the approach: recording errors occur because our termination detection and timeout logic are not sufficiently sophisticated to capture all cases. We have witnessed that it was sufficient to retry and have created a script to do so. As long as there are still errors in the output file, you can run

```
./kdo/rerun-repros.py --reports-file db/01.data
↪ --record-file db/record.1.json --path /root/repro/
↪ --outfile db/record.2.json
```

To complete the setup, we can extract from the recording information related to the reproducer execution and feed them back into the database.

```
./03.mergeRecordResults.py --infile db/01.data --outfile
↪  db/02.data --recordfile db/record.2.json
```

Concolic execution The setup for concolic execution requires creating the docker container where S2E will be downloaded, patched, compiled and executed.

```
docker build -t s2e-image .
```

This will take approximately 10-15 Minutes. We experienced in our testing a sporadic linking issue for one S2E component when compiled inside `docker`.

```
/usr/bin/ld: cannot find -lglibc-compat-main: No such file
↪  or directory
```

Re-running the exact same command a second time solves the issue. This phase executes cloning inside the docker all the subprojects of S2E, seek them to the right commit hash, patch each with our changes, and compile them. Furthermore, we install our pre-made S2E image that runs our modified kernel. We also create, for each object in the database coming from the taint verification, a new S2E project for which we create a specific configuration file that instruct our S2E plugin on how to approach the analysis. More details can be found in the provided Dockerfile.

A.3.2 Basic Test

Taint Verification We provide here the instructions on how to run a basic test for one reproducer. For instance, for our paper running example 408bf2a23593a83e9ab66cc2982c30d87e52b223 case.

```
./kdo/analysis1/run-analysis.py --record-file
↪  db/record.2.json --repro-id
↪  408bf2a23593a83e9ab66cc2982c30d87e52b223

./04.mergeReplayResults.pass1.py --infile db/02.data
↪  --outfile db/03.data --path panda/out/

./kdo/analysis2/run-analysis.py --record-file db/03.data
↪  --repro-id 408bf2a23593a83e9ab66cc2982c30d87e52b223

./05.mergeReplayResults.pass2.py --infile db/03.data
↪  --outfile db/04.data --path panda/out/

./kdo/analysis3/run-analysis.py --record-file db/04.data
↪  --repro-id 408bf2a23593a83e9ab66cc2982c30d87e52b223

./06.mergeReplayResults.pass3.py --infile db/04.data
↪  --outfile db/05.data --path panda/out/
```

It is also possible to run all the reproducers in bulk by omitting the `-repro-id` argument. That will spin up as many PANDA instances running the analysis in parallel as the machine has cores. The degree of parallelism can be controlled with `-npar`.

Concolic execution To run the concolic execution, we need to first spin and connect to the docker image that we have created during setup

```
docker run -dit --name s2e --restart unless-stopped
↪  s2e-image
docker exec -it s2e bash
```

We provide two scripts to run the tests. The first only runs the analysis on our successfully verified reproducers, as from the tables in the paper (Recommended).

```
python $DATA/scripts/run_successful_ids.py
```

The second one runs all the 222 cases from the PANDA analysis. This considerably increases the time of execution especially on small machines.

```
python $DATA/scripts/run_all.py
```

These scripts are set to run CopyKat on a medium-sized machine (20 cores and 32 GB of RAM). Based on your machine specs, you can edit under `$DATA/scripts/run_*.sh` and set how many parallel jobs–i.e., `Pool(NJOBS)`–your machine can support . We advise to set `N_CORES-1`. However, if you are heavily memory constrained (e.g., <16GB) do not exceed 10 jobs.

It is also possible to run each individual test by hand. For instance, if we want to re-run our running example from the paper:

```
cd 408bf2a23593a83e9ab66cc2982c30d87e52b223
./launch.sh
```

A.4 Evaluation workflow

Taint Verification To evaluate the results of the taint verification phase, the evaluator can inspect the final output file from the latest command. Specifically, `db/05.data` in the example above should include all the information collected on the Panda recording. It is possible to verify such output with the one provided under

```
concolic_execution_with_corruption/s2e-data/database/panda.data
```

which is the complete database for all the objects we have created during our evaluation. `panda.data` is quite large and therefore we suggest the use of file editor that can handle large files (e.g., vim). Alternatively, it is possible to verify the entry related to each reproducer against the one used during Concolic Execution. For instance, to verify the entry of our running example outputted by PANDA, it should match the file:

```
408bf2a23593a83e9ab66cc2982c30d87e52b223/
  408bf2a23593a83e9ab66cc2982c30d87e52b223.output
```

inside the S2E docker container.

Concolic Execution To verify the results of our characterization phase through concolic execution, it is possible to run from another shell connected to the S2E docker container the following commands:

```
find -name report.txt | xargs -I {} grep -l true {}
find -name report.txt | xargs -I {} grep -l true {} | wc
↪ -l
```

These commands can be used to monitor the progress of successfully characterized reproducers. The first returns a list of paths to the successful reports. The second returns just the total number. It is also possible to run the following two commands to generate a sql database and to query it to generate the results we included in our tables in the paper:

```
python $DATA/scripts/process_result.py
bash $DATA/scripts/query_table_final.sh
```

It is possible to directly verify the output of the `query_table_final.sh` command with the tables available in the paper. These commands can be re-run whenever new tests successfully terminate.

Finally, it is possible to inspect individually the various output files of each run. Each project folder will contain a folder `s2e-last` which point to the latest run. This folder contains the following files:

`debug.txt` provides the output of our plugin.

`report.txt` provides general information (e.g., number of constraints, primitive signal etc.) about the test, generated if the analysis does not fail. It can be compared with the one provided in our results on Zenodo, under the archive `constraints.zip`.

`counterfeit-*.txt` provide path-constraints collected at the memory operation or return for the memory area we characterize.

A.4.1 Major Claims

(C1): *Characterization of CCO data-only amplifier objects.*
(C2): *An automated tool, CopyKat, to find and extract characteristics of a CCO.*
(C3): *Identification of 122 CCOs, with 8 exploits demonstrating their practical use.*

A.4.2 Experiments

We designed two experiments to support our claims **C2** and **C3**. We provide below a high level description of the steps for each experiment. For more detailed instructions, please refer to commands listed above, or under the `README`s provided in the Github repository for each stage.

(E1): *[Generate PANDA DB] [10 human-minutes + 24-48 compute-hour + 50GB disk]: This experiment aims to verify the lightweight taint detected during fuzzing and generate a database with detailed information on which elements compose the CCO primitive detected.*
Preparation: Extract the reproducer metadata. Compile the reproducers that should be tested. Create the PANDA image through the docker-compose command. Generate at least one valid execution recording to be replayed in the next phase.
Execution: On the valid recording, invoke the three main PANDA passes in sequence. Integrate after each analysis pass the new output inside the DB through the corresponding merge pass.
Results: The results of this analysis are the latest output files under the folder db. These files would be the input to the concolic execution phase, and can be verified against our `panda.data` db.
To be noticed, our cache name detection is done statically and might not be correct. Whenever we were not able to detect any cache information we fill the entry with `_kmalloc_`. For the cases for which we provide exploits, we manually verified such information and update them accordingly. This can be easily done through `gdb` by calculating the size of the object type under analysis using the provided kernel image. This means that some cache name might differ from the one presented in the paper.
For simplicity and completeness, we rely on our `panda.data` database to run the next step and not the one generated in this test.

(E2): *[Concolic Execution] [1 human-Minute + 1 to 10 compute-hour + 150GB disk]: This experiment runs the validated reproducers under concolic execution and verifies that the primitive is valid after simulated corruption. It provides as output the path-constraints describing the primitive. By default as input it takes the pre-computed PANDA database that we provide. For the sake of reproducing our results, it is not necessary to run/complete the PANDA analysis to test this step.*
Preparation: Create the s2e-image through the `docker build` command.
Execution: Run and connect to the generate s2e-image. Run either manually (`./launch.sh`) or through bulk run (`run_successful_ids.py`) the reproducers. Based on the computation power available, please adjust the Pool number within the script to tailor towards the evaluator's machine. Currently, the script timeouts after 1h. This is to guarantee that the analysis does not get stuck on problematic cases and completes the highest number within a short time. Some cases might require multiple hours or attempts to successfully complete.
Results: The results of this analysis are in the `$hash$/s2e-last` folders. By using the scripts provided, aggregate the results and compare against the paper tables. While the majority ($\sim$106) reproducers should finish within a couple of hours (depending of the computational power). Some take significantly longer and many attempts to successfully be characterize. This

is due to few factors described in the paper like concurrency, inaccurate information gathered at the PANDA stage, or S2E spuriously crashing under certain conditions. These results can be verified against Tables 4, 5 and 6 in the main paper.

A.5 Notes on Reusability

Our artifact is meant to be re-used with minimal changes. The most obvious re-use opportunity is to explore CCOs under another kernel version. This would only require to instrument through our compiler pass a new kernel version, and re-generate the QEMU images and snapshots that are at the base of Syzkaller, PANDA and S2E. This is necessary to guarantee transferrability of information across the analysis (e.g., offsets, functions).

Another quite obvious re-use opportunity of CopyKat concerns the extension to other data-only object patterns. Currently, CopyKat searches and verifies only for CCOs. However, the more data-only objects are discovered, the more patterns can be encoded in out pipeline. This would require changes in how the taint is managed during execution across the fuzzing campaign and the PANDA analysis. Depending on the type of object, changes to our characterization step might be required as well.

You can include in this section any sort of instruction that you believe would help others re-use your artifact, like, for example, scaling down/up certain components of your artifact, working on different kinds of input or data-set, customizing the behavior replacing a specific module/algorithm, etc.

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/usenixsec2026/.