USENIX Security '25 Artifact Appendix: GDMA: Fully Automated DMA Rehosting via Iterative Type Overlays

Tobias Scharnowski, Simeon Hoffmann, Moritz Bley, Simon Wörner, Daniel Klischies[1], Felix Buchmann, Nils Ole Tippenhauer, Thorsten Holz, and Marius Muench[2]

CISPA Helmholtz Center for Information Security
[1]Ruhr-Universität Bochum, [2]University of Birmingham

A Artifact Appendix

A.1 Abstract

This artifact contains the information necessary to reproduce all claims from the paper *GDMA: Fully Automated DMA Rehosting via Iterative Type Overlays*.

A.2 Description & Requirements

This artifact is a fuzzing artifact and hence scales better the more resources it can use. In our setup, we ran each experiment 10 times, 24 hours each. We used two Intel Xeon Gold 5320 CPUs (26 physical cores and 2.20GHz each), 256 GB of RAM, and 1TB SSD storage.

For artifact evaluation, we recommend running the experiment with 26 cores, 3 times, 12 hours each. This ensures that even on moderate hardware, the artifact evaluation time period suffices. With this configuration, one full pipeline run takes eleven days. Please take this into account when inspecting this artifact. However, it is possible to split up the long-running parts over several machines to reduce runtime.

A.2.1 Requirements

All experiments were run on an Ubuntu 22.04 machine. We provide reduced fuzzing configurations to account for artifact reviewers without access to hardware clusters. However, our recommended configuration still requires eleven days of computing on a single piece of minimal hardware. Therefore, please plan accordingly.

A.2.2 Security, Privacy, and Ethical Concerns

It may happen that new bugs are found during the evaluation of this artifact. In that case, please report them responsibly.

A.2.3 How to Access

Please obtain all required files:

(A1) GDMA artifact: https://github.com/fuzzware-fuzzer/fuzzware/tree/DMA
(A2) the GDMA experiments artifact https://github.com/fuzzware-fuzzer/gdma-experiments
(A3) the DICE / P2IM reproduction archive: https://doi.org/10.5281/zenodo.15600641 (`dice_docker.tar.zstd`)

Stable URL The stable URL of the artifact is the following: https://doi.org/10.5281/zenodo.15600641.

A.2.4 Hardware Dependencies

This artifact is a fuzzing artifact and hence scales better the more resources it can use. In our setup, we ran each experiment 10 times, 24 hours each. We used two Intel Xeon Gold 5320 CPUs (26 physical cores and 2.20GHz each), 256 GB of RAM, and SSD storage.

The minimal requirements are 26 cores, 64GB RAM and 500GB disk storage. More cores can speed up the experiments, as can multiple machines.

A.2.5 Software Dependencies

Please install the following software dependencies:

```
docker curl ca-certificates patchelf texinfo
automake gcc zstd pip zip python3-venv
binutils-arm-none-eabi
```

Also, make sure that you are in the docker group that you can use docker without elevated permissions[1].

A.2.6 Benchmarks

None.

A.3 Setup

The setup is tested on Ubuntu 22.04.

[1]https://docs.docker.com/engine/install/linux-postinstall/

A.3.1 Installation

Installation of GDMA. Start by cloning the GDMA artifact and installing it (A1):

```
git clone --recurse-submodules -b DMA
https://github.com/fuzzware-fuzzer/fuzzware.git;

cd fuzzware; ./build_docker.sh
```

This performs a regular docker build. If this build completes the `GDMA` installation was successful.

Installation of GDMA experiments. Obtain the GDMA experiments artifact (A2):

```
git clone https://github.com/fuzzware-fuzzer/
gdma-experiments.git
```

Enter the scripts directory and install the requirements in a clean virtual environment.

```
cd gdma-experiments/scripts;
python3 -m venv venv && source venv/bin/activate &&
pip install -r requirements.txt
```

We expect all further commands to be executed from within the virtual environment. To finish the installation of the GDMA experiments artifact, move the three remaining archives (A3-A5) to their place as described below.

Setup IDEs. Some samples need their integrated development environments (IDEs) to successfully build. To obtain the required IDE files, follow the instructions at `gdma-experiments/src/02-extended-test-suite/targets/CY8CKIT-062-BLE/` and at `src/02-extended-test-suite/targets/LPC1837-mmio`. Move the resulting files into the `ides` directory.

Setup DICE / P2IM reproduction containers. Move the DICE / P2IM reproduction archive to `gdma-experiments/docker-saves`. This requires no unpacking:

```
mv dice_docker.tar.zstd
gdma-experiments/docker-saves/
```

Configuring the system for fuzzing. Execute the following commands to prepare the system for fuzzing with afl and fuzzware.

```
echo 512 > /proc/sys/fs/inotify/max_user_instances;
echo 524288 > /proc/sys/fs/inotify/max_user_watches;
echo core >/proc/sys/kernel/core_pattern;
cd /sys/devices/system/cpu;
echo performance | tee cpu*/cpufreq/scaling_governor
```

A.3.2 Basic Test

To test that GDMA works correctly, execute:

```
fuzzware/run_docker.sh;
fuzzware pipeline --help
```

This gives you a shell in a docker container and executes the fuzzware help. If `--dma` is among the listed options, then GDMA is set up correctly.

To verify that the eval scripts are set up correctly, execute

```
cd gdma-experiments/scripts && python cli.py
```

If this shows you an argparse-style overview of command-line options, then the scripts are setup correctly.

To confirm progress during the evaluation, inspect the folders of each experiment (e.g., `02-extended-test-suite/` or `01-dice-comparison/02-real-firmware/`). If it contains an output folder, then the fuzzer has at least started there. If the next experiment also contains an output folder, then the fuzzer has completed the previous experiment.

A.4 Evaluation Workflow

A.4.1 Major claims

In the paper, we make the following four major claims:

(M1) GDMA correctly models all Direct Memory Access (DMA) types from state of the art data sets.
(M2) GDMA provides coverage for 6x the DMA mechanisms compared to the state of the art.
(M3) GDMA improves coverage on real-world targets.
(M4) GDMA identifies 6 previously unknown bugs.
(M5) GDMA has a low false positive rate.

A.4.2 Open Science

In addition, this artifact contains the following 5 components in accordance to the open science policy.

(O1) The diverse dataset from Table 2.
(O2) The example applications firmware dataset from Figure 6.
(O3) Ground truth metadata for the DICE dataset.
(O4) A docker-based reproduction environment for DICE and P2IM.
(O5) The source code of GDMA.

A.4.3 Experiments

To show our major claims and open science contribution, we provide the following experiments (eleven days runtime on recommended machine):

(E1): Diverse data set reproduction: rebuild the diverse dataset from its sources (10 human minutes + 0.5 CPU hours + 45GB disk space (if docker export or import is used), O1)
(E2): Example applications reproduction: rebuild the example applications from its sources (10 human minutes + 0.5 CPU hours + 45GB disk space (if docker export or import is used), O2)

(E4): P2IM / DICE containerized reproduction environment: A dockerized environment to build both P2IM and DICE , and run each tool (10 human-minutes + 12h on recommended machine + 20GB disk space, O4)

(E5-1): Dice Unit tests: Comparison to state of the art on their unit test dataset (10 human-minutes + 48h on recommended machine + 40GB disk space, M1)

(E5-2): Dice Fuzzing tests: Comparison to state of the art on their fuzzing dataset (10 human minutes + 24h on recommended machine + 70GB disk space, M1)

(E6): Diverse dataset performance: Evaluation of GDMA on the diverse dataset (10 human minutes + 48h on recommended machine + 30GB disk space, M2)

(E7): Example applications performance: Evaluation of GDMA on the example applications dataset (10 human minutes + 48h on recommended machine + 70GB disk space, M3)

(E8): Finding new bugs: Evaluate if GDMA can be used to find new bugs (10 human minutes + 24h on recommended machine + 70GB disk space, M4)

(E9): False positives: Evaluate if GDMA finds false positives in P2IM unit test set or the diverse dataset (10 human minutes + 72h on recommended machine + 30GB disk space, M5)

We split the reproduction into three parts: build target reproduction, DICE / P2IM execution environment test, and GDMA evaluation. As the first step, enter the `scripts/` directory: `cd gdma-experiments/scripts`. The remainder of this section assumes that all commands are executed from within this directory.

A.4.4 Reproducing O3

We do not provide an experiment script to reproduce **O3**. Extraction of ground truth metadata was a one-time manual effort, as it involved reverse-engineering the provided firmware samples. Please find the results for DICE unit tests in `ground_truth_dice_perspective.yml`[2]. Please find DICE fuzzing test ground truth metadata in their respective directory, in the `config_generic_dma_manual.yml`[3] file.

A.4.5 Build Target Reproduction (E1-E2)

The first section of the artifact reproduction describes how to rebuild the diverse dataset and the example applications that we provide as part of the artifact.

Diverse Dataset (E1). To test the reproduction of the diverse dataset, execute the following commands:

b) Rebuild from scratch: Run

```
python cli.py build-2
```

This builds docker containers from scratch, importing the IDEs from `ides`.

After successfully rebuilding the targets, execute

```
python cli.py update-2 -pd
```

, which checks for each binary if the newly-built version differs from the existing one.

Example Applications (E2). The reproduction for the example applications follows the same idea. Execute

```
python cli.py build-3
```

for a rebuild from scratch. You can verify that all binaries built successfully, as above, by executing

```
python cli.py update-3 -pd
```

, or by removing them before the build process and counting them afterwards (there should be 10 binaries).

Using the new binaries. You can use the samples you built in the previous steps to perform the evaluation. However, to keep the overhead for you as low as possible we recommend to use the binaries we provide. IF you want to entirely verify that the evaluation works with the rebuilt binaries you can update the diverse dataset binaries and the example applications with the following commands

```
python cli.py update-2 -u;
python cli.py update-3 -u
```

. Note that this might invalidate some of the present configurations as some targets require manual tweaking of the memory maps. We advise you to first perform the evaluation with the dataset in the archive and to test out the update mechanism afterwards if you are interested to further investigate this aspect of the artifact.

A.4.6 DICE / P2im Execution Environment Test (E4)

This part builds and runs both DICE and P2IM, which is related work that GDMA was evaluated against. The targets for this part of the evaluation stem from the DICE evaluation and will hence be call DICE fuzzing tests.

This comes in two flavors: one rebuilds DICE and P2IM from scratch:

```
python cli.py run-dice
```

and creates an evaluation-ready docker container, the other imports existing docker images from `docker-saves`:

```
python cli.py run-dice -d
```

Both run DICE and P2IM on the DICE fuzzing tests afterwards.

You can modify fuzzing campaign parameters by editing `01-dice-comparison/02-real-firmware/01-evaluation/config.txt`. We provide wrapper scripts to directly execute DICE reruns from the `scripts/` directory. On the recommended machine, execute:

[2] 01-dice-comparison/DICE-results/ground_truth_dice_perspective.yml

[3] 01-dice-comparison/02-real-firmware/firmware/GPS-Receiver/config_generic_dma_manual.yml

```
./dice_12h_3runs.sh
```

Investigating the results. To investigate the results navigate to:

```
cd gdma-experiments/scripts/results/p2im_dice_results
```

This directory contains two subdirectories: `DICEFuzzBase` and `P2IMFuzzBase`, which contain the data for the respective platform. The structure of each of these subdirectories is `<sample>/<run>/`. To investigate the results of P2IM's/DICE's modeling and fuzzing please refer to the respective projects. In addition to the various peripheral modeling and fuzzing-related directories each run contains a `coverage` folder. In there is a file that contains the total number of basic blocks covered (`bbl_cnt`) and a file that contains the mapping of inputs to the basic blocks they cover (`bbl_cov`). Further each fuzzing run contains an `out.txt` file, that contains a mapping between fuzzing time and total number of basic blocks covered at that time. If a directory does not contain an `out.txt` file please refer to the troubleshooting section.

To verify, whether the results of the fuzzing run match those shown in the paper you can inspect the different `out.txt` files and compare the achieved number of basic blocks with Figure 7 in the paper.

Troubleshooting. Unfortunately, in some cases the coverage and plotting information can not be generated for all targets even though the fuzzing worked fine. This usually manifests itself with error messages such as:

```
cp: cannot stat 'p2im_dice_results/DICEFuzzBase/
Modbus/2/out.txt': No such file or directory
```

To verify whether this occurred during your evaluation you can perform the following steps:

```
cd gdma-experiments/scripts
find ./results/p2im_dice_results -name "out.txt"
| wc -l
```

The resulting number should be `11 * <runs>`, where <runs> is the number of repetitions specified through the run script (e.g., `dice_12h_3runs.sh` → <runs>=3). This "formula" is derived by multiplying the number of targets (6) by 2 (each is fuzzed twice: once with DICE and once with P2IM) and then subtracting the number of repetitions once, as one sample (MIDI-Synthesizer) does not work with P2IM, therefore, no output file is created.

If you notice that the number of generated `out.txt` files is lower than expected, we provide a script to rerun the coverage generation, based on the data from the fuzzing runs.

To this end, you can run the following commands:

```
cd gdma-experiments/scripts
cd ../01-dice-comparison/02-real-firmware/
04-coverage-only/
./rerun_cov.sh ../../../scripts/results/
p2im_dice_results/../02-results
```

Afterward, you can do the same verification from above to check whether the issue was fixed.

A.4.7 GDMA Evaluation (E5-E9)

This part reruns all 5 fuzzing campaigns and collects the results in the `scripts/results/` directory. To start the GDMA evaluation immediately with default parameters, move to the top-level directory `gdma-experiments` and execute:

```
./eval-recommended.sh
```

Optional Configuration. The evaluation setup is configurable to adapt to different availability of computing power. We provide configurations for a range of computing power. We recommend using one of the wrapper scripts to keep the manual configuration effort low. Execute one of the wrapper scripts (e.g. `rerun_experiments_12h_3runs.sh`) to use these preset configurations.

If your hardware setup is incompatible with the provided presets, you can configure the parameters of the fuzzing campaigns in the configuration file `scripts/.experiments-config.yml`. It contains 4 keys: `experiments` defines which of the 5 fuzzing campaigns you want to rerun, `cores-per-experiment` defines the number of cores you assign for the fuzzing campaigns, `fuzzing-time` the timeout for the fuzzer and `runs-per-firmware` the number of runs of each target.

Running the evaluation. If you want to perform the evaluation on recommended hardware and settings, execute:

```
./eval-recommended.sh
```

in `gdma-experiments` directory root. You can run the full evaluation with preselected parameters by executing one of the provided wrapper scripts (e.g. `rerun_experiments_12h_3runs.sh`). The scripts run all fuzzing campaigns with the parameters indicated in the wrapper name. Afterwards, it performs all evaluations, placing the results in the `scripts/results/` directory.

If you have similar or better hardware resources than those used by us in our evaluation (see A.2.4), we you can run the full evaluation as ran by us. To do that, enter the `scripts` directory and execute the script:

```
rerun_experiments_24h_10runs.sh
```

However, we understand that not everybody has access to such resources. If this is the case for you, we recommend to execute

```
rerun_experiments_12h_3runs.sh
```

from inside the `scripts` directory. This requires less than 25% of compute time, and yields comparable results. Note that the x-axis (the time scale) of the resulting plots will differ. Edit and use `python estimate_runtime.py` to estimate the runtime for a given number of execution time, runs and cores.

Investigating the results. The `scripts/results` directory contains all evaluation artifacts.

01_unit.txt (E5-1). This contains an ASCII version of Table 7 in the paper.

02_fuzzing.pdf (E5-2). This contains the coverage plots of the DICE fuzzing targets (Figure 7 in the paper). If you did not run the DICE / P2IM environment test (see A.4.2), the data for plotting DICE and P2IM is equal to that used in the paper. Running the DICE / P2IM environment test automatically overwrites the old data with the latest results.

02.txt (E6). This file contains information about password characters found in the diverse dataset. It marks a target as successfully executed if some password character were found. It requires 1 identified password character if a single DMA buffer was involved, or 2 password characters (1 per buffer) if two buffers were involved.

02-fp.txt (E9). This is the result file of diverse dataset false-positive analysis. This analysis compares the DMA configurations identified by GDMA to a manually constructed ground truth and reports differences.

03-cov.pdf (E7). The coverage plots for the example application dataset (Figure 6 in the paper).

04.txt (E8). This experiment replays all reproducer inputs to verify that they indeed crash the target. If you want to verify the CVEs found by GDMA, you need to manually inspect the crashes that your run found. Please find the raw output of the fuzzing campaign in `04-finding-new-bugs/output/`. Note that the crashes are not necessarily triggering the identified CVEs. Also, note that depending on your execution time, some bugs may not be found, as the time to crash is high for some bugs.

05.txt (E9). This experiment investigates if `dma_config.yml` files were created for any of the P2IM unit tests. The existence of any `dma_config.yml` is a false positive.

A.5 Notes on Reusability

Our artifact uses a python script as main tool to rerun our experiments. This section contains information on how to use all parts individually. Each of the following commands is invoked as

```
python cli.py <headline>
# e.g. python cli.py run-experiments
```

`python cli.py --help` gives a brief overview over each command, and each subcommand also has an individual `--help` output.

A.5.1 Running the Campaigns

run-experiments. Runs the actual fuzzing campaigns. Is configured via the `.experiments-config.yml` file. This file controls the parameters of the fuzzing run. Use the `-r` flag to actively select the configuration, otherwise the default configurations in the directories of the fuzzing run will be used. Use `-d` to perform a dry-run: it prints all the commands instead of running them.

A.5.2 Evaluating the Campaigns

eval-1-1. Builds the unit test table (Table 7 in the paper). By default, builds a human readable table (ASCII art style). Pass `--latex` to get latex code for the table (this is also used in the paper).

eval-1-2. Builds the plot for the DICE Fuzzing targets (Figure 7 in the paper). You can pass directories that contain the output from an experiment run via command line flags (use `--help` to see which ones) relative to the repository root. By default, it uses paths as present in the repository structure. Use `--runtime` and `--num-runs` to set the correct runtime and number of runs per target.

eval-2. Evaluates the functionality of the DMA emulation mechanism. This takes the results of a fuzzing campaign on the diverse data set and confirms that GDMA found password characters. If there was a single consecutive buffer in memory, the evaluation checks for the discovery of the first password character. If the DMA mechanism uses two buffers, the evaluation checks whether GDMA found password characters for both buffers. Use `--results` to pass an alternative path to the output of `run-experiments`.

eval-2-fp. Check that GDMA found the correct DMA configuration during its run. We compare the Memory-mapped I/O (MMIO) register and the written buffer pointer to compare configurations. Note that some targets have a non-deterministic way of providing a DMA channel, and consequently, multiple MMIO registers are correct. `--data` points to the results of the fuzzing campaign, `--groundtruth-root` points to a directory that contains the reference configurations.

eval-3. Build the coverage plots for the real-world targets (E3, Figure 6 in the paper). Interface is similar to **eval-1-2**.

eval-4. Rerun the bug reproducers found in the repository and investigate if they crash. Pass target directory with `--targets`.

eval-4-hooks. Rerun the reproducers found in the repository with a special configuration that prints *Heureka* when found. If you pass `-f`, a new fuzzing campaign is started that prints *Heureka* when the bugs are triggered.

eval-5. False positive analysis. Search for `dma_config.yml` in the results directory of experiment 5.

A.5.3 Rebuilding the Targets

build-2. Rebuild targets for E1. This comes in two flavors: the default is a full rebuild from scratch. This requires the presence of several proprietary IDEs which we cannot provide in the final repository (after Artifact evaluation) due to legal reasons. Please see the `readme.md` files of the individual targets

for details on how to obtain them via the official channels.The full rebuild builds docker images and runs the build process in docker, moving the final build binaries in the respective `output` folder. If you pass `-e`, then the docker image will be saved for later reuse. If you pass `-d`, docker images will be loaded from the `docker-saves` folder in the repository root (assuming you exported them before). These docker saves contain the IDE setups and will build binaries during the run.

update-2. Check if the binaries found in the `output` folders of the dataset for E2 match the ones used in the evaluation. If you pass `-pd`, a diff is printed. If you pass `-u`, binaries that are not identical will be updated (and also get a new `config_autogen.yml`).

build-3. `build-2`, but for E2 targets.

update-3. `update-2`, but for E2 targets.

run-dice. Build P2IM and DICE , the related work tools that we evaluate against. Similar to the interface of `build-2`.

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/.