USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Posthammer: Pervasive Browser-based Rowhammer Attacks with Postponed Refresh Commands

Finn de Ridder, Patrick Jattke, and Kaveh Razavi, *ETH Zurich*

https://www.usenix.org/conference/usenixsecurity25/presentation/de-ridder

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: Posthammer: Pervasive Browser-based Rowhammer Attacks with Postponed Refresh Commands

Finn de Ridder
ETH Zurich

Patrick Jattke
ETH Zurich

Kaveh Razavi
ETH Zurich

A Artifact Appendix

A.1 Abstract

Posthammer shows that browser-based Rowhammer attacks are pervasive. In particular, we show that the majority of DDR4 devices are vulnerable to *clflush*-free Rowhammer patterns that the attacker can launch from client-side JavaScript.

Posthammer includes the following artifacts: first, an experiment for inducing and measuring refresh postponement. Second, a fuzzer for triggering Rowhammer bit flips natively using *clflush*-free and *refresh-postponing* patterns in a large search space. Third, a JavaScript version of the fuzzer that searches for effective patterns in a reduced search space. Fourth, an exploit that obtains an arbitrary read-write primitive in the address space of the JavaScript process.

A.2 Description & Requirements

To reproduce our results, the evaluator needs:

1. The hard- and software dependencies listed under A.2.3 and A.2.4.
2. The artifacts themselves, which are available at `https://doi.org/10.5281/zenodo.14738152` and `https://github.com/comsec-group/posthammer` (preferred).

After unpacking the artifacts, execute the following commands:

1. For the first two artifacts, the experiment and native fuzzer, in `./native-fuzzer`, execute `./main.sh |& tee dump`. Simply run it again if you get an error about `mem.o` missing. The refresh postponement experiment requires the `SPLIT_DETECT` macro to be defined.
2. For the JavaScript fuzzer, in `./js-dbg-hugepages`, execute `make clean && make; make`. We run `make` twice to force execution despite the transpiler warnings.
3. For the exploit, in `./js-exploit`, also execute `make clean && make; make`.

The exploit may fail either due to a segfault or an assertion failing (e.g. because it cannot find an eviction set). In these cases, please try again. Similarly, the native fuzzer may fail to find its first eviction set, but should otherwise not crash.

The exploit will run until it has (intentionally) segfaulted at `0x1337` while the native fuzzer will fuzz indefinitely while writing its results to `./pattern/flip.csv`. Human-friendly output can be found in `dump` (if captured as suggested above). Search for `->` to view the bit flips that have been triggered.

A.2.1 Security, privacy, and ethical concerns

The artifacts are safe. As mentioned above, upon successful execution, the exploit causes a harmless segmentation fault at address `0x1337`. This segmentation fault will be reported in the kernel log, see `dmesg`. Unsuccessful runs will trigger arbitrary but equally benign segmentation faults in the address space of the script.

A.2.2 How to access

The latest version of the artifacts is available at `https://github.com/comsec-group/posthammer`. The directory structure suggests where which artifact can be found: the exploit is contained in `js-exploit`, the native fuzzer in `native-fuzzer`, and the JavaScript fuzzer in `js-dbg-hugepages`.

A.2.3 Hardware dependencies

1. A desktop machine with an Intel Core i7-7700K (Kaby Lake) processor. *This is a strict requirement for the exploit.* The native fuzzer also works on Intel Core i7-8700K (Coffee Lake) CPUs.
2. A (single) vulnerable DDR4 DIMM. The specifics of the DIMMs used in the paper are given in the paper's appendix. For testing, we recommend a Samsung (A) DIMM, as they are most vulnerable to Posthammer.

A.2.4 Software dependencies

1. A Debian-based operating system. We have used Ubuntu 18.04.6 under Linux 5.4.0-150-generic. Although we recommend to use exactly these versions, newer ones might also work. For example, the native fuzzer also works on Ubuntu 22.04.5 under Linux 5.15.0-130-generic.
2. Certain software packages. The details are given in the README. The installation should be straightforward.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

We refer the evaluator to the README for the installation instructions.

A.3.2 Basic Test

The following tests can be used to verify the environment.

1. The availability and version of the TypeScript transpiler:

```
$ tsc --version
Version 2.7.2
```

2. The availability and version of the JavaScript shell:

```
$ ./jsshell-130/js --version
JavaScript-C130.0
```

A.4 Evaluation workflow

A.4.1 Major Claims

C0: Rowhammer patterns crafted as specified in the paper (Sections 5 and 6) induce refresh postponement. This is proven by Figure 2 of the paper, which can be reproduced using E0, see below.

C1: On the majority of DDR4 devices, these non-uniform and/or refresh postponing *clflush*-free Rowhammer patterns trigger bit flips while the self-evicting patterns used in previous work do not. This is proven by Experiment 6 (Table 2) in the paper.

C2: It is possible to trigger these bit flips from JavaScript with a reduced search space. See also Experiment 6 and Table 2.

C3: The bit flips triggered by these patterns can be used to obtain an arbitrary read-write primitive in the JavaScript runtime.

A.4.2 Experiments

The experiments below map linearly to the claims in A.4.1.

E0: *Refresh postponement*: produces Figure 2 of the paper and therefore shows that our patterns induce refresh postponement.

1. Navigate to `./native-fuzzer/pattern/pattern.c`. Open the file and enable the `SPLIT_DETECT` macro.
2. Execute `./native-fuzzer/main.sh`. This should take around 30 minutes. Plot the data it has written to `./native-fuzzer/split.csv`.

E1: *Native fuzzer:* explores the *clflush*-free, non-uniform, and refresh-postponing pattern space. Triggers bit flips on most DDR4 devices, see again Table 2.

1. Make sure the `SPLIT_DETECT` macro is *undefined* (default).
2. Execute `./native-fuzzer/main.sh |& tee dump`. Depending on the vulnerability of the DIMM, it may take several hours until the first bit flip. As Table 2 shows, however, for most DIMMs, 6 hours should suffice.

As mentioned A.2, we recommend piping the output to a file and grepping it for the arrow symbol `->` to check for bit flips.

E2: *JavaScript fuzzer*: shows that the native patterns translate to JavaScript. Relies on huge pages for convenience.

1. Navigate to `./js-dbg-hugepages` and execute `make clean && make; make`.

The `Makefile` should automatically enable transparent huge pages (THPs).

E3: *Exploit*: the exploit. Does *not* rely on huge pages. Uses two bit flips to obtain an arbitrary read-write primitive in the JavaScript runtime. To showcase the primitive, we write to virtual address `0x1337` and segfault.

1. Navigate to `./js-exploit` and execute `make clean && make; make`.

The exploit may take up to an hour to complete. Moreover, it may segfault before completion due to *unwanted* bit flips. It will print `About to segfault at 0x1337...` just before segfaulting as planned, which may be verified by inspecting `dmesg`.

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/.