Ultimate SLH: Taking Speculative Load Hardening to the Next Level

Zhiyuan Zhang, *The University of Adelaide;* Gilles Barthe, *MPI-SP and IMDEA Software Institute;* Chitchanok Chuengsatiansup, *The University of Melbourne;* Peter Schwabe, *MPI-SP and Radboud University;* Yuval Yarom, *The University of Adelaide*

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

USENIX'23 Artifact Appendix: Ultimate SLH: Taking Speculative Load Hardening to the Next Level

Zhiyuan Zhang ⚜, Gilles Barthe ◎ 🄸ᵈᵉᵃ, Chitchanok Chuengsatiansup ꩜,
Peter Schwabe ◎ ⚜, Yuval Yarom ⚜

⚜ The University of Adelaide, Adelaide, Australia
◎ MPI-SP, Bochum, Germany
🄸ᵈᵉᵃ IMDEA Software Institute, Madrid, Spain
꩜ The University of Melbourne, Melbourne, Australia
⚜ Radboud University, Nijmegen, The Netherlands

A   Artifact Appendix

A.1   Abstract

We provide the artifact of USLH in a GitHub repository. The artifact includes the PoC of leaking secrets from resolving branch conditions and variable-time instructions. The artifact includes a real word example of how LLVM-SLH fails to protect the OpenSSL library. Besides the demonstration of vulnerabilities, the artifact also includes a fix to SLH and a gadget searching tool implemented in LLVM.

A.2   Description & Requirements

A.2.1   Security, Privacy, and Ethical Concerns

Running artifact does not need a root privilege. All data fed to the program are randomly generated. The provided code does not access files other than those described in the README. The artifact evaluation involves compiling the Clang and OpenSSL source code. Please follow the instructions and do not install these software; otherwise they may disturb the system wide configuration of Clang and OpenSSL.

A.2.2   How to Access

The artifact and documentation are available on GitHub: https://github.com/0xADE1A1DE/USLH/tree/e23d4292723b11fa56efb9c237b6db201be97bfa.

The source code for the USLH implementation of SLH is available at https://doi.org/10.5281/zenodo.7704637.

A.2.3   Hardware Dependencies

A machine with an Intel processor (8th Gen, 9th Gen, 10th Gen) running Ubuntu (not virtual machine) is necessary. The artifact has been tested on processor i7-10710U, running Ubuntu 20.04. To build the customized compiler, your machine has to have at least 8GB RAM.

A.2.4   Software Dependencies

The artifacts requires a GCC compiler to compile Clang.

A.2.5   Benchmarks

None.

A.3   Set-up

A.3.1   Installation

You will need to download and compile LLVM, but you do not need to install it. Instructions on building Clang is available at https://clang.llvm.org/get_started.html. Note that building Clang with Release version is sufficient for the artifact evaluation. You can find more instructions at README under the folder USLH/LLVM_FIX.

You will need to download and compile OpenSSL-1.1.1q, which is available at https://www.openssl.org/source/old/1.1.1/. Instructions on compiling OpenSSL with customized compiler and flags are available at the README in the folder USLH/PoC/openssl_leakage.

A.3.2   Basic Test

To evaluate the fix to LLVM-SLH, having a working customized Clang is necessary. After compiling the Clang, you should check if Clang is properly compiled by compiling a program with *clang* under $path_to_folder/build/bin/.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): Resolving branch conditions leaks secret if the secret resides in branch operands (E1, E4).
(C2): USLH prevents leakages from resolving branches by hardening branch conditions (E1, E4).
(C3): Variable-timing instructions leak secret by checking the cache status of a secret-independent memory. Specifically, when a sequence of floating point instructions is fed with a *slow value*, the secret-independent memory access may not be scheduled to execute (E2, E3).
(C4): USLH mitigates the vulnerability of variable-timing instructions by hardening the operands of variable-timing instructions. (E3)
(C5): We provide a LLVM backend pass to find potential gadgets (E5).

A.4.2 Experiments

(E1): [1/60 human-minutes, 1/3600 cpu-hour]: Leak secret from resolving the branch condition and fix it. A detailed instruction is available in README under the folder USLH/PoC/condition.
Preparation: You need to have the USLH compiled to mitigate the vulnerability. Please refer to README in the folder *USLH/LLVM_FIX* for more instructions.
Execution: You need to modify the *folder* in *compile.bash* to compile the program with and without fix to LLVM-SLH. You then run the executable file with a parameter, which is either 1 or 0.
Results: When executing *leak*, if the fed value is 1, the program should return a measurement with cache miss penalty. If the fed value is 0, the program should return a measurement with cache hit in most cases. When executing *fix*, no matter what value fed is, the program should always return a measurement with cache miss penalty.
(E2): [1 human-minutes 1/60 cpu-hour] Blocking reservation station under speculation. The code is available at USLH/PoC/test_rs_limit. README contains detailed instructions.
Preparation: None.
Execution: Run the command *python3 test.py $max $min* to test how many pairs of *sqrtsd, mulsd* can block the RS during the speculation. You need to change *val* in run.bash to test *fast value* ot *slow value*.
Results: For *slow value*, with fewer pairs of floating-point operations, the secret-independent memory will not be accessed during the speculation. The actual number of pairs is various from processors. On 11th and 12th Gen Intel processors, you may not see the effect as they have larger ROB and RS.
(E3): [1/60 human-minutes, 1/3600 cpu-hour] Leak secret from variable-timing instructions. The code is avail-

able at USLH/PoC/variable_time. README contains detailed instructions.
Preparation: You need to complete the last experiment and adjust the number of floating-point instructions manually. Note that you may want to reduce the number of pairs in this experiment as the vulnerable function is slightly different from the one in E2.
Execution: Execute the program with or without mitigation with *attack.bash* or *mitigate.bash*.
Results: The program processes a secret value bit-by-bit. It returns the eight measurements of accessing the secret-independent memory, the guessed secret and whether the guess is correct or not.
(E4): [1/60 human-minutes, 1/3600 cpu-hour] Leak secret from *BN_mul_word* in OpenSSL. The code is available at USLH/PoC/openssl_leakage. README contains detailed instructions.
Preparation: You need to install the OpenSSL with the customized Clang. Please refer to the README file for more instructions on how to compile OpenSSL with a customized compiler and flags.
Execution: Execute the program with *./crun $val* where *val* is either 1 or 0.
Results: When *val* is 0, the measurement should be cache hit; otherwise it should return a cache miss penalty. By fixing the OpenSSL with USLH, no matter what *val* is, it should always return cache miss penalty.
(E5): [Heavily dependent on processors and targets] Find gadgets. The code is available at USLH/LLVM_FIX. README contains detailed instructions.
Preparation: You need to have a compiled USLH.
Execution: Compile the program that you have interest with command *$path_to_binary/clang file -mllvm -x86-mir-analyze*
Results: If there is a gadget, the terminal prints *Found it -> function_name*. Then you need to refer the source code to review the code.

A.5 Notes on Reusability

USLH improves the LLVM-SLH by hardening more vulnerable operands or instructions. You can use the customized compiler to build safer programs. To play with different functionalities of USLH, you can enable features with command '-mllvm -x86-slh-xxx'. The LLVM backend pass performs static analysis on machine IR. You can use it to find potential leakages. To use it, you need to compile the program with command *-mllvm -x86-mir-analyze*.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at

https://secartifacts.github.io/usenixsec2023/.