ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

USENIX Security '25 Artifact Appendix: SCASE: Automated Secret Recovery via Side-Channel-Assisted Symbolic Execution

Daniel Weber
CISPA Helmholtz Center for Information Security

Lukas Gerlach
CISPA Helmholtz Center for Information Security

Leon Trampert
CISPA Helmholtz Center for Information Security

Youheng Lü
SCHUTZWERK GmbH

Jo Van Bulck
DistriNet, KU Leuven

Michael Schwarz
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This paper proposes SCASE, a novel methodology for inferring secrets from an opaque victim binary using symbolic execution guided by a concrete side-channel trace. Our key innovation is utilizing the memory accesses observed in the side-channel trace to prune the symbolic-execution space, thus avoiding state explosion. To demonstrate the effectiveness of our approach, we introduce Athena, a proof-of-concept framework to recover secrets from Intel SGX enclaves via controlled channels automatically. We show that Athena can automatically recover the 2048-bit secret key of an enclave running RSA and the 256-bit key from an RC4 KSA implementation. Furthermore, we demonstrate key recovery of OpenSSL's 256-bit AES S-Box implementation and recover the inputs to OpenSSL's binary extended Euclidean algorithm. To demonstrate the versatility of our approach beyond cryptographic applications, we further recover the input to a poker-hand evaluator. Our findings indicate that constraining symbolic execution via side-channel traces is an effective way to automate software-based side-channel attacks without requiring an in-depth understanding of the victim application.

A.2 Description & Requirements

The majority of our artifact only requires a Linux installation supporting Python 3 and a recent *angr*[1] installation. For recovering the 2048-bit RSA key, a machine with at least 64 GB of RAM is required. The only (optional) hardware requirement is required to regenerate the memory traces for the 2048-bit RSA key. For regenerating the memory traces for the 2048-bit RSA key, which we consider an optional experiment as we provide the traces alongside our artifact, an Intel SGX machine is required.

[1]https://github.com/angr/angr

A.2.1 Security, privacy, and ethical concerns

Our framework does not pose any security, privacy, or ethical concerns, as it only executes Python code. Nevertheless, the (not required) execution of our *ptrace tracer* requires the user to disable ASLR. For this, scripts exist to dis- and enable ASLR in the corresponding folder. Furthermore, our artifact contains real-world code used as meaningful victim applications. While the source code of these applications is publicly available, we cannot guarantee that the code is free of security, privacy, or ethical concerns. To the best of our knowledge, the code does not contain any such concerns.

A.2.2 How to access

The artifact is publically available on GitHub
`https://github.com/cispa/scase`.
Furthermore, the artifact is archived on Zenodo with DOI `https://doi.org/10.5281/zenodo.15609410`. Note that, when fetched from Zenodo, the file `scase-traces-*.zip` needs to be extracted to `./traces`.

A.2.3 Hardware dependencies

Our artifact requires a Linux machine with at least 62 GB of allocatable RAM. The optional regeneration of the memory traces for the RSA key requires an Intel SGX machine.

A.2.4 Software dependencies

Our installation instructions are written for Ubuntu 22.04, but the artifact should work on most recent Linux distributions. We require Python 3.10 or higher, an *angr* installation, and at least 62 GB of allocatable RAM.

A.2.5 Benchmarks

Our artifact contains the memory traces used during the evaluation of our framework. These memory traces are CSV files containing the memory accesses of the victim application. While we provide instructions on regenerating the memory traces, users are free to use the provided memory traces.

A.3 Set-up

A.3.1 Installation

After cloning the repository, create a virtual environment and install the required dependencies described under "Dependencies" in the `athena/README.md` file.

A.3.2 Basic Test

To run a basic test, execute the following command (after loading the virtual environment and from within the folder `./athena`): `python3 ./hex_elf.py`. Upon successful execution, the output should contain `Encoded solution: 8`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Athena can automatically recover secrets for varying granularities of memory traces, which are not recovered without memory trace guidance. This is shown by Experiment E1 (Section 5.1.1) and Experiment E6 (Sections 5.1.2 and 5.2).

(C2): Athena can recover an AES key from OpenSSL's S-Box implementation and the input of OpenSSL's binary extended Euclidean algorithm implementation. This is shown by Experiment E2 (Section 5.3) and Experiment E3 (Section 5.4).

(C3): Athena can recover the input to an RC4 KSA implementation. This is shown by Experiment E4 (Section 5.5).

(C4): Athena can recover the input to non-cryptographic applications, as demonstrated by its applicability to the TPT hand evaluator. This is shown by Experiment E5 (Section 5.6).

A.4.2 Experiments

To ease the reproduction of our results, we provide precompiled victim applications and memory traces in the `traces` folder. The Athena framework uses angr to perform symbolic execution on these binaries. Thus, the scripts, also referred to as *Athena wrappers* in the latter, never actually execute these binaries but rather emulate their execution. Experiment E7 (optional) describes how to recompile the victim applications and regenerate the memory traces. Note that the regeneration of the traces requires an Intel SGX machine with SGX-Step[2] installed, which is not required for the other experiments.

[2] https://github.com/jovanbulck/sgx-step

(E1): [Jump-Table Example] *[30 human-minutes + 2 compute-hours]:* This experiment shows Athena recovering the key from our Jump-table toy victim.
How to: The Athena wrapper for this can be found in `./athena/hex_elf_eval.py`. Our results, which are used in Figures 5 and 6 of the paper, can be found in `traces/jump-table/`. The victim program's code can be found in `./victim-programs/jump-table/`
Preparation: Traverse to the folder `./athena` and load the installation's venv.
Execution: Execute `python3 ./hex_elf_eval.py` and `python3 ./hex_elf_eval_noprune.py`. This generates subfolders with the following structure: `eval_data_g_<bytelength>_<option>`. When `<option>` equals `noprune`, the symbolic execution is not pruned by the side-channel trace.
Results: The resulting statistics are contained in the file `statistics.csv` for each subfolder. These (accumulated) numbers correspond to the paper's Figures 5 and 6. The most noteworthy result is that the `<...>_noprune` subfolders should indicate that the value could not be recovered. This can be confirmed by checking the column `incorrect_bytes` in the corresponding `statistics.csv` file.

(E2): [AES S-Box] *[30 human-minutes + 30 compute-minutes]:* This experiment shows Athena recovering the symmetric key from OpenSSL's AES S-Box implementation.
How to: The Athena wrapper for this can be found in `./athena/aes_openssl.py`. The victim program's code can be found in `./victim-programs/openssl-aes-sbox/`.
Preparation: Traverse to the folder `./athena` and load the installation's venv.
Execution: Execute `python3 ./aes_openssl.py`. This recovers the 256-bit AES key and prints whether the key is correct. The key can be verified by checking the file `./victim-programs/openssl-aes-sbox/victim.c`. The command `python3 ./aes_openssl_eval.py` can be used to execute the experiment for different memory trace granularities.
Results: The execution of `aes_openssl.py` should print that the key was successfully recovered. The execution of `aes_openssl_eval.py` creates two folders `eval_data_g01` and `eval_data_g01_dfonly`, where the latter restricts the guidance to the data-flow trace. Both folders contain a file `statistics.csv`, which contains the detailed statistics (similar to E1). The number of incorrect bytes for e should align with the paper's Table 1.

(E3): [BEEA] *[30 human-minutes + 1 compute-hour]:* This experiment shows Athena recovering the key from OpenSSL's binary extended Euclidean algorithm

(BEEA) implementation.

How to: The Athena wrapper for this can be found in `./athena/beea_openssl.py`. The victim program's code can be found in `./victim-programs/openssl-beea/`.

Preparation: Traverse to the folder `./athena` and load the installation's venv.

Execution: The experiment can be executed by running `python3 ./beea_openssl.py`. This recovers the second argument to `BN_gcd`. Note that this takes around 10-14 hours to finish.

Results: The execution of `aes_openssl.py` should print the key contained in the variable `p_min_one` from `./victim-programs/openssl-beea/main.c`.

(E4): [RC4-KSA] *[30 human-minutes + 5 compute-minutes]:* This experiment shows Athena can recover the RC4 key from a key scheduling algorithm (KSA) implementation.

How to: The Athena wrapper for this can be found in `./athena/rc4_elf.py`. The victim program's code can be found in `./victim-programs/rc4-ksa/`.

Preparation: Traverse to the folder `./athena` and load the installation's venv.

Execution: Execute `python3 ./rc4_elf.py`. This recovers the key from `victim-programs/rc4-ksa/main.c`.

Results: The execution of `rc4_elf.py` should print the key recovered from the binary. This key should match with the content of `./victim-programs/rc4-ksa/key.hex`.

(E5): [TPT Hand Evaluator] *[30 human-minutes + 2 compute-minutes]:* This experiment shows Athena can recover secrets from non-cryptographic applications, in this case, the input to a poker-hand evaluator.

How to: The Athena wrapper for this can be found in `./athena/poker_elf.py`. The victim program's code can be found in `./victim-programs/tpt-hand-evaluator/`.

Preparation: Traverse to the folder `./athena` and load the installation's venv. Additionally, unpack the binaries contained in the tar balls in `./traces/tpt-hand-evaluator/`. For this, run `find . -name '*.tar.gz' -exec tar -xzvf {} \;` from within the folder.

Execution: The experiment can be executed by running `python3 ./poker_elf.py`. This recovers the card array from `victim-programs/tpt-hand-evaluator/test.c`.

Results: The execution of `poker_elf.py` should print the array contained in the variable `cards` in `./victim-programs/tpt-hand-evaluator/test.c`. As we executed this program multiple times with varying inputs, the folder `./traces/tpt-hand-evaluator/` does not only contain the binaries and memory traces but also the expected outcomes for each input. The combinations of input and expected output are enumerated. For example, the files `cftrace1.csv`, `dftrace1.csv`, `victim1`, and `solution1.txt` belong together.

(E6): [Intel SGX Square+Multiply Enclave] *[2 human-hours + 10 compute-minutes]:* This experiment recovers a 2048-bit key from an Intel SGX enclave implementation of a toy Square+Multiply algorithm.

How to: The Athena wrapper for this can be found in `./athena/sm_enclave.py`. The victim program's code can be found in `./victim-programs/square-multiply-enclave/`.

Preparation: Traverse to the folder `./athena` and load the virtual environment from the installation.

Execution: The experiment can be executed by running `python3 ./sm_enclave.py`.

Results: The printed key should match the variable `secret` contained in the file `square-multiply-enclave/encl.c`.

(E7 - Optional): [Victim Recompilation and Trace Regeneration] *[4 human-hours + 2 compute-hours]:* This experiment is about recompiling the victim applications and regenerating the memory traces provided in `./traces/`.

How to: For each victim application, the corresponding folder contains a `README.md` file with complication instructions, e.g., the instructions to recompile the Jump-Table example are found in `./victim-programs/jump-table/README.md`.

Preparation: The Jump-Table example does not require any special preparation besides a `gcc` installation. The AES and BEEA examples require the corresponding OpenSSL library to be compiled. The hand evaluator requires a `g++` installation. The Square+Multiply Enclave requires Intel SGX and SGX-Step installed.

Execution: Follow the instructions in `victim-programs/<victim>/README.md` to recompile the victim application and regenerate the memory traces.

Results: The binaries and traces should match the ones provided in `./traces/<victim>`. Note that due to varying environments, it is unlikely that the binaries and resulting traces are identical. One can copy the resulting binaries and memory traces to the corresponding folder in `./traces/` rerun the corresponding experiment.

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/`.