

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Prezta: Provable Remote Execution of Zero-Trust Authorization using SNARKs

Zhongjing Wei, Osaid Muhammad Ameer, Yupeng Zhang, and Nikita Borisov, *University of Illinois Urbana-Champaign*

https://www.usenix.org/conference/usenixsecurity26/presentation/wei-zhongjing

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology



USENIX Security '26 Artifact Appendix: Prezta: Provable Remote Execution of Zero-Trust Authorization using SNARKs

Zhongjing Wei, Osaid Muhammad Ameer, Nikita Borisov, Yupeng Zhang
University of Illinois Urbana-Champaign

A Artifact Appendix

A.1 Abstract

This artifact contains the source code and benchmarks for Prezta, a SNARK-based authorization architecture for Operational Technology (OT) systems. The artifact consists of two main components: (1) the Prezta prototype implementation using RISC Zero zkVM v3.0.3, including proof generation and verification; (2) an XACML-to-Rust compiler that translates XACML 3.0 access control policies into Rust code compatible with zkVM execution. The artifact allows reproducing all main experimental results reported in Section 5 of the paper, including end-to-end performance (Figures 1–2), ablation study (Table 2), and user cycles breakdown (Table 3).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. The artifact is a research prototype that does not perform any destructive or privacy-violating operations.

A.2.2 How to access

The artifact is permanently archived on Zenodo (concept DOI, always resolving to the latest version):

https://doi.org/10.5281/zenodo.20303461

The repository is also available on GitHub at https://github.com/walotta/ZK_Zero_Trust. The main experiments (E1, E2) use the `master` branch. The ablation study (E3, E4) requires the `AblationStudy` branch, which is included in the repository.

A.2.3 Hardware dependencies

An x86_64 Linux machine with at least 32 GB RAM is required to reproduce the experiments. The paper's results were obtained on an AWS c7i.4xlarge instance (16 vCPUs, Intel Xeon Scalable, 32 GB RAM). No GPU is required; CUDA support is optional and can accelerate proof generation.

A.2.4 Software dependencies

- Ubuntu 22.04+ (or equivalent Linux distribution)
- System packages: `build-essential`, `pkg-config`, `libssl-dev`, `git`, `curl`
- Rust stable toolchain (via `rustup`)
- RISC Zero zkVM v3.0.3 (installed via `rzup`)
- Python 3.10+ with `pip` (packages: `lxml`, `Jinja2`, `cryptography`, `tqdm`, `maturin`)
- Nix (optional, provides reproducible environment via `flake.nix`)

A.2.5 Benchmarks

The XACML 3.0 conformance test suite is included in the repository under `tools/xacml-to-rust/policy_test_set/`. Running the compiler (E0) generates 368 policy code files with corresponding request/response pairs into `jwt_zkvm_testing/` (with RSA) and `zkvm_testing_without_rsa/` (without RSA). Pre-signed JWT tokens are stored in `test_jwts/` and automatically included in the generated datasets.

A.3 Set-up

A.3.1 Installation

```
# System dependencies (Ubuntu)
sudo apt update && sudo apt install -y \
  build-essential pkg-config libssl-dev \
  git curl python3-venv

# Install Rust toolchain
curl --proto '=https' --tlsv1.2 -sSf \
  https://sh.rustup.rs | sh -s -- -y
source "$HOME/.cargo/env"

# Install RISC Zero toolchain (pin to v3.0.3)
curl -L https://risczero.com/install | bash
source "$HOME/.bashrc"  # or restart shell
rzup install cargo-risczero 3.0.3
```

```
rzup install r0vm 3.0.3

# Clone main repository
git clone \
  https://github.com/walotta/ZK_Zero_Trust.git
cd ZK_Zero_Trust
mkdir -p logs

# Clone the XACML-to-Rust compiler
git clone \
  https://github.com/osaidameer/xacml-to-rust.git \
  tools/xacml-to-rust

# Install compiler dependencies
python3 -m venv ~/.venv
source ~/.venv/bin/activate
pip install lxml Jinja2 cryptography tqdm maturin

# Build regex compiler extension
cd tools/xacml-to-rust/compile_regex
PYO3_USE_ABI3_FORWARD_COMPATIBILITY=1 \
  maturin develop --features python
cd ~/ZK_Zero_Trust
```

A.3.2 Basic Test

After running E0 (see Section A.4), run the batch script on a single policy to verify the environment is correctly configured. First, temporarily limit execution to one test case:

```
sed -i '36 a testcases_names = ["IIA001"]' \
  batch_exec.py
python3 batch_exec.py
# Revert the modification
git checkout -- batch_exec.py
```

Expected: the log in `logs/` should contain `Total cycles: 262144`, `Gen time elapsed: ~28s`, and `Verify time elapsed: ~15ms`. The first run includes full compilation and may take 5–10 minutes. Completion is indicated by the message `All policies processed.` on stdout.

Note: We recommend running all long experiments inside `tmux` or `screen` to prevent SSH disconnections from interrupting execution.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Prezta's XACML compiler supports 83% of the XACML 3.0 conformance suite (322 out of 389 testable policies). This is proven by experiment (E1) and reported in Section 5.1.

(C2): Prezta's proof generation completes in ∼14–28 seconds per policy on a c7i.4xlarge instance, and verification takes 14–16 ms. This is proven by experiment (E1) and reported in Section 5.1.

(C3): Without RSA verification, 88.6% of policies achieve proof generation in ∼7 seconds. This is proven by experiment (E2) and reported in Figure 2.

(C4): Prezta's optimizations (RSA precompile + regex DFA pre-compilation) reduce prover time by 47.7× compared to a naive approach. This is proven by experiment (E3) and reported in Table 2.

(C5): The user cycles breakdown shows RSA contributes 45% and regex 47% of user cycles. This is proven by experiment (E4) and reported in Table 3.

A.4.2 Experiments

(E0): XACML-to-Rust Compilation [2 human-min + 2 compute-min]: Run the XACML-to-Rust compiler to generate Rust policy code from XACML sources. This step produces the datasets used by experiments E1 and E2.
Preparation: Ensure the Python virtual environment is activated (`source ~/.venv/bin/activate`).
Execution: `cd ~/ZK_Zero_Trust`
`bash run_compiler.sh true`
`bash run_compiler.sh false`
Results: Expected: ∼363 policies compiled successfully out of 389 attempted per configuration. The 26 failures are due to unsupported XACML higher-order functions (not used in practice). The generated datasets are placed directly in the directories that `batch_exec.py` reads from, so experiments E1 and E2 can be run immediately after this step.

(E1): End-to-End Performance [5 human-min + 2.5 compute-hours]: Run all conformance test policies with full Prezta (RSA + regex optimizations).
Preparation: Ensure E0 has been run with `bash run_compiler.sh true`. Ensure you are on the `master` branch.
Execution: `python3 batch_exec.py`
Completion is indicated by `All policies processed.` on stdout.
Results: The log is written to `logs/no_rsa_batch_<timestamp>.log`. The summary line reports total test cases and failures. Expected: 368 cases processed, ∼46 failures. The conformance suite contains 389 testable policies; 373 compile to Rust successfully; 368 have complete test inputs (request/response/JWT); 322 produce correct authorization decisions, yielding a coverage rate of 322/389 ≈ 83%. Note: 8 of these 322 involve malformed requests that fail deserialization; since the subject cannot furnish the required attributes, the correct decision is Deny (no proof needed), which aligns with the XACML conformance expectation. Extract metrics via:
`cp logs/<logfile> data/end2end/end2end.log`
`python3 data/end2end/extract.py \`

```
  -o data/end2end/metrics.csv
```
The resulting `metrics.csv` contains per-policy prover time (∼28s or ∼14s), verification time (14–16 ms), and proof size (238–250 KB), corresponding to Figures 1 and the text in Section 5.1.

(E2): End-to-End without RSA [5 human-min + 50 compute-min]: Run all policies with RSA verification removed to reproduce Figure 2.

Preparation: Ensure E0 has been run with `bash run_compiler.sh false`. On the `master` branch, change the dataset path in `batch_exec.py`:

```
sed -i 's/jwt_zkvm_testing/\
  zkvm_testing_without_rsa/' batch_exec.py
```

Execution: `rm -rf target/`

```
python3 batch_exec.py
```

Results: Expected: ∼89% of policies complete in <10s (∼7.3s), with total cycles of 65,536 (2^{16}). This matches the paper's claim that 88.6% drop to 7 seconds. After recording results, revert the modification:

```
git checkout -- batch_exec.py
```

(E3): Ablation Study [15 human-min + 35 compute-min]: Reproduce Table 2 by running policy IIC057 under three optimization configurations.

Preparation: Switch to the `AblationStudy` branch:

```
git checkout AblationStudy
```

The file `methods/guest/src/main.rs` contains three configuration constants at lines 16–18. The batch script is pre-configured to run only IIC057. The test input files are:

```
REQ=tools/xacml-to-rust/jwt_zkvm_testing/\
  requests/Policy_IIC057.json
RESP=tools/xacml-to-rust/jwt_zkvm_testing/\
  responses/Policy_IIC057.json
JWT=tools/xacml-to-rust/jwt_zkvm_testing/\
  jwts/Policy_IIC057.json
```

Execution: Run three configurations sequentially:

(a) "None" (naive, no optimizations): In `methods/guest/Cargo.toml`, comment out the `rsa` and `sha2` lines under `[patch.crates-io]`. In `methods/guest/src/main.rs`, set:

```
const USE_STD_REGEX: bool = true;
const BYPASS_REGEX: bool = false;
const BYPASS_RSA: bool = false;
```

Then run:

```
rm -rf target/
cargo run --release -- $REQ $RESP $JWT
```

Expected: ∼10.5M user cycles, ∼1300s (∼22 min).

(b) "+RSA" (RSA precompile only): Uncomment the `rsa` and `sha2` patches in `Cargo.toml`. Keep `USE_STD_REGEX=true`, `BYPASS_RSA=false`.

```
rm -rf target/
cargo run --release -- $REQ $RESP $JWT
```

Expected: ∼805K user cycles, ∼176s.

(c) "+Regex" (full Prezta): Set `USE_STD_REGEX=false`, `BYPASS_RSA=false`.

```
rm -rf target/
cargo run --release -- $REQ $RESP $JWT
```

Expected: ∼153K user cycles, ∼28s.

Results: Compare user cycles, prover time, verifier time, and proof size against Table 2. User cycles should match within 1%; prover times may vary by ∼10% due to machine load.

(E4): User Cycles Breakdown [5 human-min + 10 compute-min]: Reproduce Table 3 by isolating each module's contribution.

Preparation: On the `AblationStudy` branch, ensure `[patch.crates-io]` patches are active (uncommented).

Execution: Run IIC057 with both RSA and regex bypassed:

```
# In methods/guest/src/main.rs, set:
# const USE_STD_REGEX: bool = false;
# const BYPASS_REGEX: bool = true;
# const BYPASS_RSA: bool = true;
rm -rf target/
cargo run --release -- $REQ $RESP $JWT
```

Results: Expected user cycles ∼13,507 ("Others"). Combined with E2's IIC057 result (∼84,855 = Others + Regex) and E3c (∼153,513 = full), derive:

- Others = 13,507
- Regex = 84,855 − 13,507 = 71,348
- RSA = 153,513 − 84,855 = 68,658

These match Table 3 (paper: 13,508 / 71,429 / 68,598).

A.5 Notes on Reusability

The XACML-to-Rust compiler (`tools/xacml-to-rust/`) can be used to compile custom XACML policies for zkVM execution. New policies can be tested by placing the generated Rust code in `methods/guest/src/main.rs` and running the host binary. The system can also be adapted to other zkVMs (e.g., SP1, Zisk) by replacing the RISC Zero-specific APIs with equivalent calls.

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/.