# Zero-Knowledge Middleboxes

Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish, *NYU*

https://www.usenix.org/conference/usenixsecurity22/presentation/grubbs

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

# A Artifact Appendix

## A.1 Abstract

The purpose of this artifact is to allow reproduction of the performance results in Section 8, specifically the channel opening microbenchmark table (Figure 6) and the full proof generation benchmark for the case studies (Figure 7). All runtime estimates in this abstract are for a Linux system with an 8-core 2.2 GHz AMD EPYC 7571 CPU and 32 GB of RAM. All of our code is available on GitHub.

After installing dependencies, which should take roughly ten minutes, reproducing these results has two steps: (1) circuit generation and (2) proof generation. Circuit generation takes as input a (roughly) human-readable programmatic description of our circuits written in an extension of Java, and outputs a gate-level description of the corresponding arithmetic circuit. This programmatic circuit description is an intermediate representation obtained by partially compiling the original handwritten xJsnark source code. We do not require the original xJsnark source for the artifact evaluation—reading it requires installing a specific version of a large and unwieldy IDE called MPS—but our GitHub repository includes instructions on viewing the xJsnark source.

Circuit generation involves heavily optimizing the circuit description, and so is computationally quite expensive, and will take up to twenty minutes (to generate the nine example circuits in this artifact). The purpose of re-running the circuit generation as part of the artifact is to allow users to reproduce the claimed gate counts for our circuits. We provide a single script to automatically perform all of circuit generation.

After the circuits' descriptions have been generated, the last step is proof generation. Proof generation takes as input the circuit descriptions as well as sample circuit inputs (e.g., TLS handshake transcripts and ciphertexts), generates public parameters, produces proofs, and verifies them. The provided proof generation script outputs information about the time taken to generate and verify proofs, as well as the sizes of the public parameters. We estimate this will take in total up to twenty minutes (to complete all nine circuits in the artifact).

## A.2 Artifact check-list (meta-information)

- **Algorithm:** zkSNARKs, Groth16
- **Program:** xJsnark, libsnark
- **Compilation:** Java, cmake
- **Data set:** Manually generated test data. Included.
- **Run-time environment:** Ubuntu 20.04, OpenJDK 11.0.13
- **Hardware:** 32 GB RAM, 8 cores
- **Metrics:** Circuit size, proving time, verification time, parameter size
- **Experiments:** Bash scripts
- **How much disk space required (approximately)?:** 3 GB
- **How much time is needed to prepare workflow (approximately)?:** 10 min
- **How much time is needed to complete experiments (approximately)?:** 40 min
- **Publicly available:** GitHub: https://github.com/pag-crypto/zkmbs/
- **Archived (stable URL):** https://github.com/pag-crypto/zkmbs/tree/096ed18772d8e63f4a03e7f4d16e118aa3923135

## A.3 Description

### A.3.1 How to access

Our artifact's code is publicly available on GitHub here: https://github.com/pag-crypto/zkmbs

This appendix contains all the instructions specific to installation and reproducing the paper's benchmarks.

### A.3.2 Hardware dependencies

We recommend using a machine with 8 cores and at least 32 GB RAM.

### A.3.3 Software dependencies

The only major dependency is Java. We recommend using a GNU/Linux system and have provided installation scripts compatible with the Ubuntu 20.04 Linux distribution.

## A.4 Installation

1. Clone the git repository and change to the root directory (time required: < 1 minute):

```
$ git clone https://github.com/pag-crypto/zkmbs.git
$ cd zkmbs/
```

2. Install jsnark (a library used by xJsnark) and its dependencies by running this script inside `zkmbs/` (time required: 5–10 minutes): `$ ./install_deps_jsnark`

    - If you can't use the script, follow the "jsnark installation instructions" here: https://github.com/akosba/jsnark#prerequisites
    - On some systems, this step may fail when trying to install the dependencies of libsnark as specified in this file: https://github.com/akosba/libsnark/blob/213547311d16644bde7ef806b77dfae25c7f734c/.gitmodules. Please edit all URLs in your local version of the file at `zkmbs/jsnark/libsnark/.gitmodules` (which should be cloned by this point) to use `https` (and not `git`) and try again.

3. Enter `gen/` and compile xJsnark: `$ cd gen/` and `$ ./compile_circuits`. The exact output will depend on the system but it should finish without any errors. On Ubuntu, our output looks like this:

   ```
   Note: Some input files use ...
   Note: Recompile with -Xlint:unchecked ...
   compilation SUCCESS
   ```

## A.5 Experiment workflow

After installation, the structure of the main directories should look like this:

```
zkmbs
    +-- gen
    |     +-- circuits
    |     +-- logs
    |     +-- src
    +-- jsnark
```

The experiment scripts will be run inside `gen/`. The Java source code describing the circuits is located in `gen/src/`.

Experiment 1 will generate full circuits from these descriptions and store them in `gen/circuits/`. Experiment 2 will use these circuit descriptions to generate public parameters and measure proving and verification times using sample input files located in `gen/`.

## A.6 Evaluation and expected results

The main performance claims in our paper are stated in Figures 6 and 7. There are nine circuits involved in our experiments (the five entries of Figure 6 and the four entries of Figure 7). The first experiment reproduces the "Total" columns of the two tables. The second experiment reproduces the "Time" and "SRS" columns while ensuring that verification time is under 5 ms. We recommend using a system with at least 32 GB RAM as generating proofs for the largest of our circuits (ChannelBaseline) requires a lot of heap space, and in fact causes errors on systems with just 16 GB memory.

Note that both Figures 6 and 7 list per-subcircuit gate counts that sum to the "Total" count. Our code only allows verifying the gate counts of the entire circuit, as the per-subcircuit counts were approximated by manually inspecting the functions used to build each circuit.

**Experiment 1: Reproduce Gate Counts:** The aim is to generate circuits from our descriptions and reproduce the total gate counts of each circuit (the **Total** columns of the two tables). This experiment can be repeated by running the script `./reproduce_total_counts` (time required: 20 minutes) in the `gen/` directory. The script outputs into file `column_total.txt`, which should look like this after the script finishes:

```
ChannelBaseline              747.9    # BCO
ChannelShortcut              111.1    # SCO
Channel0RTT                   60.7    # ECO
ChannelAmortized              19.1    # ACO^AES
ChannelAmortized_ChaCha        8.7    # ACO^Cha
Firewall_HS                  150.1    # Firewall
DNS_Amortized_ChaCha          17.6    # DoT
DNS_Amortized_doh_get         48.1    # DoH GET
ODOH_Amortized                48.1    # ODoH
```

Note that the "# ..." are added here to map to the abbreviations used in Figures 6 and 7. The numbers obtained should be very close to the ones above with perhaps slight variation coming from the performance of xJsnark's optimizer on different systems. Some of the values shown here are different than that of the "Total" columns in Figures 6 and 7 as those were rounded for presentation.

**Experiment 2: Reproduce Times and SRS:** The aim is to reproduce the structured reference string sizes (**SRS** columns), proving (**Time** columns) and verification time (always under 5 ms) for each circuit. This experiment can be repeated by running the script `./reproduce_times_srs` (time required: 20 minutes) inside the `gen/` directory.

The script outputs into file `columns_ptime_srs_vtime.txt`, the contents of which after a sample execution are as follows:

```
ChannelBaseline           92.7 s   1179 MB    2.6 ms
ChannelShortcut           15.6 s    148 MB    1.6 ms
Channel0RTT                8.4 s     79 MB    1.6 ms
ChannelAmortized           2.9 s     26 MB    1.7 ms
ChannelAmortized_ChaCha    1.4 s     13 MB    1.6 ms
Firewall_HS               21.2 s    206 MB    1.6 ms
DNS_Amortized_ChaCha       3.1 s     29 MB    2.1 ms
DNS_Amortized_doh_get      6.8 s     72 MB    2.6 ms
ODOH_Amortized             7.9 s     76 MB    2.6 ms
```

Proof generation is a randomized algorithm; the results reported in the paper are the median of five runs. We have observed variations of up to 15% for proving time and 2 ms for verifier time, in either direction. The script above performs just one run per circuit.

## A.7 Experiment customization

We provide two additional scripts to reproduce the above benchmarks for an individual circuit: `./generate_circuit DNS_Amortized_ChaCha` and `./prove_and_verify DNS_Amortized_ChaCha`, where "`DNS_Amortized_ChaCha`" can be replaced with any of the nine circuits.

## A.8 Notes

**Custom Inputs.** As the circuit metrics we evaluate (gate counts, parameters sizes and running times) are independent of the actual input used to generate the proofs, input customization isn't required to reproduce our results. The experiments generate valid proofs using fixed input files (`test.txt, test_doh.txt, test_wildcard.txt`) provided in the `gen/` directory. These files contain sample data extracted from a real TLS 1.3 connection and a Merkle tree blocklist of two million entries. We provide instructions in our GitHub repository on generating sample data from new DNS requests and custom Merkle trees.

**Editing Circuit Descriptions.** Our experiments generate circuits using the Java files in the `gen/src/` directory. These are in turn generated from xJsnark's custom language files that are editable only with an IDE called MPS. To inspect and edit our circuits, we recommend installing the MPS IDE by following the instructions here in our GitHub repository: `https://github.com/pag-crypto/zkmbs#installation-instructions-mps`.

## A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.