USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

# OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries

Pratap Singh, *Carnegie Mellon University;* Joshua Gancher, *Northeastern University;* Bryan Parno, *Carnegie Mellon University*

https://www.usenix.org/conference/usenixsecurity25/presentation/singh

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: OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries

Pratap Singh
Carnegie Mellon University

Joshua Gancher
Northeastern University

Bryan Parno
Carnegie Mellon University

## A Artifact Appendix

### A.1 Abstract

**Paper** Cryptographic security protocols, such as TLS or WireGuard, form the foundation of a secure Internet; hence, a long line of research has shown how to formally verify their high-level designs. Unfortunately, these formal guarantees have not yet reached real-world implementations of these protocols, which still rely on testing and ad-hoc manual audits for security and correctness. This gap may be explained, in part, by the substantial performance and/or development overhead imposed by prior efforts to verify implementations.

To make it more practical to deploy verified implementations of security protocols, we present OwlC, the first fully automated, security-preserving compiler for verified, high-performance implementations of security protocols. From a high-level protocol specification proven computationally secure in the Owl language, OwlC emits an efficient, interoperable, side-channel resistant Rust library that is automatically formally verified to be correct.

We produce verified libraries for all previously written Owl protocols, and we also evaluate OwlC on two new verified case studies: WireGuard and Hybrid Public-Key Encryption (HPKE). Our verified implementations interoperate with existing implementations and their performance matches unverified industrial baselines on end-to-end benchmarks.

**Artifact** This artifact includes the Haskell source code of OwlC, as well as the various case studies described in the paper: 14 toy protocols in Owl, an example of a verified application built on top of an Owl protocol, and our performant, interoperable implementations of WireGuard and HPKE. We also provide a Docker container in which to run the evaluations described in our paper, and scripts to automate those evaluations.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

OwlC is a security-preserving compiler for cryptographic protocols in the Owl language; thus, using OwlC does not entail direct security, privacy or ethical concerns. Our artifact does not contain any attacks or exploits, nor is it intended to be destructive in any manner. To provide further isolation to users, we provide a Dockerfile which will build a self-contained Docker image with all dependencies for our evaluation.

#### A.2.2 How to access

Our artifact is available on Zenodo at `https://doi.org/10.5281/zenodo.15605318`. We provide a zip file `owl-usenix2025-aeval.zip` containing all files required for the evaluation.

#### A.2.3 Hardware dependencies

Our artifact requires an x86-64 machine with at least 32 GB of RAM and 30 GB of free disk space.

#### A.2.4 Software dependencies

The artifact has been tested on Ubuntu 20.04 and Docker version 24.0.5. More recent versions, or other operating systems, may work as well. We require Python 3 (version 3.8.10 or newer) and `pip3` on the host device, with the packages `matplotlib`, `numpy`, and `prettytable` available; these will be installed during the Set-up phase (A.3). For (**E2**), we require the following flags to Docker, to set up a WireGuard VPN tunnel between two containers:

```
--cap-add NET_ADMIN --device \
    /dev/net/tun:/dev/net/tun
```

All experiments require Internet access to download Haskell, Rust, or Go dependencies.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

1. Ensure that Docker, Python3, and `pip3` are installed and available on the host system.

2. Download and unzip the artifact archive. All remaining steps should be conducted at the top level of the resulting directory `owl-usenix2025-aeval`.

3. Install required Python dependencies:

   ```
   pip3 install -r requirements.txt
   ```

   If desired, the Python dependencies can be installed in a virtual environment such as `venv`.

4. Build the Docker image:

   ```
   docker build -t owlc-aeval .
   ```

   Note that this may take up to 30 minutes, and requires Internet access to download dependencies. Alternatively, a pre-built Docker image is available on the Docker Hub at https://hub.docker.com/repository/docker/ps1729/owlc-aeval/. To pull the image and tag it as `owlc-aeval`, as expected by the scripts:

   ```
   docker pull ps1729/owlc-aeval
   docker tag ps1729/owlc-aeval owlc-aeval
   ```

While our artifact has been developed and tested for an x86-64 Linux machine, other architectures may work as well. To build the image on Apple Silicon, pass the `--platform linux/amd64` argument to `docker build`, as:

```
docker build --platform=linux/amd64 -t \
    owlc-aeval .
```

Note that this will use Rosetta emulation to run the scripts, which will likely incur large overheads and result in very different performance measurements.

### A.3.2 Basic Test

We provide a helper script, `run_owlc.sh`, that runs OwlC on a specified file, then verifies the generated library using Verus. The library is generated in `extraction/src/lib.rs`. As a simple test, run:

```
./run_owlc.sh owl_toy_protocols/dhke.owl
```

This will typecheck, compile, and verify our Diffie-Hellman case study. It should take about 2 minutes the first time (subsequent runs may be faster). The script will print `Done!` at the end if everything was successful; it will also print `verification results:: 105 verified, 0 errors` or similar near the end of the output, confirming that Verus verified the generated code.

## A.4 Evaluation workflow

Note that some of our evaluations and claims refer to the extended version of our paper, available on ePrint at https://ia.cr/2025/1092. The extended version contains three extra appendices, including Appendices E and F that contain additional details on our WireGuard benchmarks, but is otherwise identical to our submitted version.

### A.4.1 Major Claims

**(C1):** *OwlC is a security-preserving compiler for protocols in the Owl language, producing Rust libraries verified with Verus to be free of explicit and implicit information leakage.* This claim is evaluated in Figure 13 and Section 10 of our paper.

**(C2):** *OwlC's Rust protocol libraries for WireGuard and HPKE have performance equivalent to state-of-the-art industrial implementations.* This claim is evaluated in Sections 10.2 and 10.3 of our paper, and Appendices E and F of our extended version.

### A.4.2 Experiments

Note: You should run all of the `.sh` scripts **on your host machine**; you don't need to spin up the Docker container in interactive mode to run them. The scripts will themselves spin up containers as necessary to run the experiments. All scripts should be run from the top-level directory `owl-usenix2025-aeval`.

**(E1):** *[Run OwlC on all examples] [5 human-minutes + 20 compute-minutes]:*
This experiment substantiates claim (**C1**) by running OwlC on all sixteen examples from our paper, then using Verus to verify each of the resulting implementations. It should produce a table similar to Figure 13 in our paper.
**How to:** Run the following command:

```
./run_owlc_on_all.sh
```

**Results:** Once complete, the script should print a table with the same structure as Figure 13 in our paper. Note that some of the line counts for Verus may differ slightly due to changes in OwlC, and the verification times may be shorter due to improvements in Verus since the paper was accepted. The raw data will be saved to `run_owlc_on_all.csv`, and a copy of the formatted table will be saved to `run_owlc_on_all.txt`.
If any of the table rows contain ERROR, that indicates that either Owl failed to typecheck the file, OwlC failed to compile it, or Verus failed to verify the generated code.

**(E2):** *[Benchmark WireGuard end-to-end performance] [10 human-minutes + 4 compute-hours]:*
This experiment substantiates claim (**C2**) by measuring the end-to-end performance of several WireGuard implementations based on OwlC's generated WireGuard library, compared against industrial baselines. For details on how we constructed our OwlC-based WireGuard implementations, see Section 9.2.2, and for details on the baselines and comparisons we performed, see Section 10.2.1 and Appendix E in our extended version.
**How to:** Run the following command:

```
./bench_wireguard_end_to_end.sh
```

This script calls the Python scripts `bench_wg_linedelay.py` and `bench_wg_mss.py` with arguments to run all benchmarks. Since this benchmark needs to orchestrate two Docker containers connected by a Docker network bridge, we run the Python scripts on the host machine.

**Results:** Once complete, this script will generate four graphs in both `.png` and `.pdf` format, as follows:

- `bench_wg_linedelay-go.{png|pdf}`: Should be comparable to Figure 14 in our paper.
- `bench_wg_linedelay-rs.{png|pdf}`: Should be comparable to Figure 17 in our extended version.
- `bench_wg_mss-go.{png|pdf}`: Should be comparable to Figure 16 in our extended version.
- `bench_wg_mss-rs.{png|pdf}`: Should be comparable to Figure 18 in our extended version.

In all cases, the generated graphs should be comparable in terms of the relative performance of the three WireGuard implementations in question; the precise measured throughput values may differ from our paper due to different hardware and Docker overhead. The raw data for each graph is saved in the corresponding `.csv` files, and in formatted tables in the corresponding `.txt` files.

**(E3):** *[Benchmark HPKE end-to-end performance] [10 human-minutes + 5 compute-minutes]:* This experiment substantiates claim (**C2**) by measuring the end-to-end performance of our HPKE implementation based on OwlC's generated HPKE library, compared against an industrial baseline. For details on how we constructed our OwlC-based HPKE implementation, see Section 9.3.2, and for details on the baselines and comparisons we performed, see Section 10.3.

**How to:** Run the following command:

```
./bench_hpke.sh
```

**Results:** Once complete, this script will generate two text tables, printed to the shell and saved to `bench_hpke.txt`. The raw data will be saved to `bench_hpke.csv`. The contents of the tables should be comparable to Figure 15 in our paper in terms of relative performance; in particular, the percentage differences from the baselines in each table should be comparable.

**(E4):** *[Micro-benchmark WireGuard handshake] [10 human-minutes + 5 compute-minutes]:*
This experiment substantiates claim (**C2**) by measuring the performance of OwlC's generated routines for the WireGuard handshake in isolation. For discussion of the micro-benchmark, see Section 10.2.2, and for details, see Appendix F in our extended version.

**How to:** Run the following command:

```
./microbench_wg_handshake.sh
```

**Results:** Once complete, this script will generate two text tables, printed to the shell and saved to `microbench_wg_handshake.txt`. The raw data will be saved to `microbench_wg_handshake.csv`. The contents of the tables should be comparable to Figure 19 in our extended version in terms of relative performance; in particular, the percentage differences from the baselines in each table should be comparable.

**(E5):** *[Micro-benchmark WireGuard transport] [10 human-minutes + 45 compute-minutes]:*
This experiment substantiates claim (**C2**) by measuring the performance of OwlC's generated routines for the WireGuard transport layer in isolation. For discussion of the micro-benchmark, see Section 10.2.2, and for details, see Appendix F in our extended version.

**How to:** Run the following command:

```
./microbench_wg_transport.sh
```

**Results:** Once complete, this script will generate a graph in `microbench_wg_transport.{pdf|png}`. The raw data will be saved in `microbench_wg_transport.csv`, and in a formatted table in `microbench_wg_transport.txt`. This graph should be comparable to Figure 20 in our extended version in terms of the relative performance of the three routines under test. Note that the generated graph will not have the data series for $\text{OwlC}_{\text{unopt}}$: that corresponds to an old, unoptimized version of OwlC, about which we do not make any performance claims.

## A.5 Notes on Reusability

The `run_owlc.sh` script provides a convenient way to run Owl and OwlC on a protocol. OwlC generates a library crate in `extraction/`, with the generated code in `extraction/src/lib.rs`. The other files in `extraction/src/` are handwritten Verus definitions for the generated code; of particular interest may be `speclib.rs`, which contains our definitions of the `ITree` (Section 4.1), `DeclassifyingOp` (Section 5.2), and corresponding token structures (Sections 4.2 and 5.2). In `execlib.rs`, `mod secret` defines our `SecretBuf` abstraction (Section 5.1).

OwlC is under active development as part of the Owl project. The open-source repository is at https://github.com/secure-foundations/owl.

### A.5.1 Echo Server case study

As discussed in Section 7, we developed a simple case study of a verified echo server, available in the `echo_server_example` directory. The directory `echo_server_example/extraction` contains a copy of the generated protocol library, with the addition of `server.rs`, a handwritten file containing a verified Verus implementation of the main echo server logic (Section 7.2). To verify the echo server example code with Verus, run the script

`verify_echo_server_example.sh` on your host machine (not inside the Docker container).

The echo server illustrates how to connect a library generated by OwlC to a larger verified codebase, by hand-writing ITree specifications for the application logic. Verus will verify that the application logic follows its handwritten ITree specification. However, as discussed in Section 7, Verus does not presently feature information-flow control or cryptographic reasoning, so it cannot prove cryptographic security properties about handwritten ITrees. Developers must therefore take care that any handwritten ITrees do not leak protocol secrets or otherwise compromise the security properties of the code proven in Owl. The best practice for developing verified applications using OwlC is to write as much of the application logic as possible in Owl, with handwritten ITrees only for simple interfaces and functionality that does not directly interact with secrets.

### A.5.2 Incorporating code generated by OwlC into larger Rust projects

OwlC generates a library crate that can straightforwardly be integrated into verified code in Verus. Examples of this are in the `echo_server_example` directory, as we discuss below. Our WireGuard and HPKE case studies illustrate how to integrate the generated libraries into pre-existing Rust codebases. We discuss some high-level considerations here.

Integrating a library generated by OwlC into a pre-existing Rust codebase requires building an interface between verified and unverified code. As with any such interface, the developer must make sure that the unverified calling context satisfies the preconditions of the verified routine. For OwlC, this means that all arguments to generated protocol routines must match the Owl types of the corresponding Owl protocol routine. That is, if an Owl routine takes as argument a key $k$, then the generated Verus routine must only be called with the runtime value of the key $k$.

OwlC relies on Rust abstract types and Verus verification conditions to maintain important soundness and correctness properties. Our ITree tokens (Section 4.2) cannot be created or duplicated, so that implementation code cannot forge tokens to justify incorrect I/O operations. Our `SecretBuf` wrapper types (Section 5.1) are opaque, so that implementation code cannot leak secrets via side channels. However, connecting to unverified Rust code may necessitate breaking some of these abstraction barriers. For instance, an application may need to reveal a `SecretBuf` to display a decrypted plaintext message to the user.

When connecting to unverified code, developers may therefore need to add "escape hatches" that enable such functionality. This must be done with extreme care. We recommend that any such escape hatches be annotated with a Verus precondition of `requires false`, so that verified code cannot use them, and the generated libraries remain sound.

## 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/usenixsec2025/.