USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

A Tale of Two Worlds, a Formal Story of WireGuard Hybridization

Pascal Lafourcade and Dhekra Mahmoud, *Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Mines Saint-Etienne, LIMOS, 63000 Clermont-Ferrand, France;* Sylvain Ruhault and Abdul Rahman Taleb, *Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI), France*

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

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: A Tale of Two Worlds, a Formal Story of WireGuard Hybridization

Pascal Lafourcade[1], Dhekra Mahmoud[1], Sylvain Ruhault[2] and Abdul Rahman Taleb[2]

[1]Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Mines Saint-Etienne, LIMOS, 63000 Clermont-Ferrand, France
[2]Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI), France

A Artifact Appendix

A.1 Abstract

These artifacts accompany research paper "A Tale of Two Worlds, a Formal Story of WireGuard Hybridization". They allow to reproduce all results described in this paper, concerning symbolic evaluation of security properties and implementation of protocols related to hybridization of WireGuard. These artifacts are composed of two folders. The first folder **artifacts_implementation** concerns our Rust implementation of WireGuard, PQ-WireGuard⋆ and Hybrid-WireGuard. The second folder **artifacts_evaluation** concerns our symbolic analysis of WireGuard (with fix for anonymity based on psk), PQ-WireGuard, PQ-WireGuard⋆ and Hybrid-WireGuard, with the help of TAMARIN, PROVERIF and DEEPSEC verifiers.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Our artifacts are available on a permanent public repository[1]. To access them, execute (MacOS may require to remove \):

```
record_url=$(curl -Ls -o /dev/null -w '%{url_effective}' \
https://doi.org/10.5281/zenodo.15551056)
curl -L -o artifacts-usenix-2025-wireguard-hybridization.zip \
"$record_url/files/artifacts-usenix-2025-wireguard-hybridization."\
"zip?download=1"
unzip artifacts-usenix-2025-wireguard-hybridization
cd artifacts-usenix-2025-wireguard-hybridization
```

A.2.3 Hardware dependencies

Our artifacts do not require any specific hardware. All results are reproducible on a standard laptop with 12 cores of CPU 1.8 GHz and 16 Go of RAM.

[1] https://doi.org/10.5281/zenodo.15551056

A.2.4 Software dependencies

Our artifacts require the use of Docker Engine[2]. For our tests, we used Docker Engine version 28.2.2 and we successfully tested Docker deployment on both Linux Ubuntu 24.04.2 LTS and MacOS Sequoia 15.4. We use TAMARIN release version 1.10.0, Maude version 3.5, PROVERIF version 2.05, DEEPSEC version 2.0.2, Python version 3.12.3, Package sympy, GNU parallel, Clang version 18.1.3 and Rust version 1.87.0. All these dependencies are installed automatically when building Dockerfile (subsubsection A.3.1).

A.2.5 Benchmarks

Our artifacts allow to reproduce similar benchmarks as the one given in our paper (Table 9), to compare the efficiency of WireGuard, PQ-WireGuard⋆ and Hybrid-WireGuard. Our benchmarks focus on the construction of InitHello message by Initiator, and on the processing of InitHello and construction of RespHello by Responder. Execution time is averaged on a chosen number of executions of these operations, that is a parameter of the benchmark (subsubsection A.4.2).

A.3 Set-up

A.3.1 Installation

Before building Docker, it is adviced to perform a complete cleanup of Docker environment. For this, execute (warning: this cleanup stops and deletes *all* Docker containers, images, volumes, networks):

```
sh run_docker_clean.sh
```

Then, execute:

```
docker buildx build --platform linux/x86_64 -t hyb-wg .
```

This should build the Docker image with all dependencies (subsubsection A.2.4). Docker build takes around 30 minutes.

[2] https://docs.docker.com/engine/install

A.3.2 Basic Test

To check Docker image, execute:

```
docker run -it hyb-wg
```

Then inside Docker, execute:

```
sh run_basic_test.sh
```

This should launch, successively, TAMARIN, PROVERIF, DEEPSEC verifiers and Cargo Rust, hence their respective versions shall successively appear (subsubsection A.2.4). Furthermore, this basic test launches evaluations of trace properties (resistance against unilateral and bilateral unknown key share attacks, session uniqueness, agreement and secrecy, including perfect forward secrecy) for PQ-WireGuard* protocol. A set of CNFs shall be printed, as in Table 2, equal to the ones described in Table 4 of our paper, for PQ-WireGuard* protocol. This basic test shall finish in less than 2 minutes.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We propose a Rust implementation of protocols WireGuard, PQ-WireGuard*, Hybrid-WireGuard. This is proven by the experiment (**E1**) described in Section 7.2 of our paper, with results reported in Table 9 of our paper.

(C2): We propose, with TAMARIN, PROVERIF and DEEPSEC verifiers, symbolic proofs of trace properties (session uniqueness, resistance against UKS attacks, message agreement and secrecy, including PFS) and observational equivalence properties (anonymity, strong secrecy), for WireGuard (with fix for anonymity based on psk), PQ-WireGuard, PQ-WireGuard* and Hybrid-WireGuard. This is proven by experiments (**E2**), (**E3**), (**E4**), described in Sections 4, 5, 6 of our paper, whose results are reported in Table 4 of our paper.

A.4.2 Experiments

(E1): *[Efficiency evaluation] [1 human-minutes + 10 compute-minutes]: based on Rust implementation, build an application that mesures efficiency of construction of InitHello message by Initiator, and processing of InitHello and construction of RespHello message by Responder.*
Preparation: Inside Docker image (subsubsection A.3.2), in folder `artifacts_implementation`, file `run_all.sh` contains line `cargo run - -b 100`. Value `100` is the chosen number for the benchmarks, as described in subsubsection A.2.5.
Execution: Run `sh run_all.sh`.
Results: Printed results shall be as in Table 1, where number of executions is the one chosen for benchmarks (subsubsection A.2.5), message sizes correspond to Table 8 of our paper and timings correspond to Table 9 of our paper.

Table 1: Results for Experiment (**E1**).

```
Average time over 100 executions

WireGuard:
InitHello message size: 196 bytes
RespHello message size: 140 bytes
InitHello construction time: 0.230 ms (std = 0.061 ms)
InitHello consumption time + RespHello construction time: 0.507 ms (
      std = 0.148 ms)

PQ-WireGuard*: (static kem: Classic-McEliece-460896, ephemeral kem:
      ML-KEM-512)
InitHello message size: 1124 bytes
RespHello message size: 1032 bytes
InitHello construction time: 0.326 ms (std = 0.168 ms)
InitHello consumption time + RespHello construction time: 0.612 ms (
      std = 0.148 ms)

Hybrid-WireGuard: (static kem: Classic-McEliece-460896, ephemeral kem
      : ML-KEM-512)
InitHello message size: 1156 bytes
RespHello message size: 1064 bytes
InitHello construction time: 0.481 ms (std = 0.074 ms)
InitHello consumption time + RespHello construction time: 1.211 ms (
      std = 0.178 ms)
```

(E2): *[Trace properties evaluation] [1 human-minutes + 4 compute-hours]: based on* SAPIC$^+$ *files, generate all* PROVERIF *and* TAMARIN *files and evaluate trace properties (session uniqueness, resistance against UKS attacks, message agreement and secrecy, including PFS) for WireGuard (with fix for anonymity based on* psk*), PQ-WireGuard, PQ-WireGuard*, Hybrid-WireGuard.*
Preparation: Inside Docker image (subsubsection A.3.2), in folder `artifacts_evaluation`, file `run_trace_properties.sh` contains all necessary commands that allows to, sequentially, perform all evaluations. These use PROVERIF verifier, for all protocols, and TAMARIN verifier for WireGuard (with fix for anonymity based on psk) and Hybrid-WireGuard. We did not use TAMARIN verifier for PQ-WireGuard and PQ-WireGuard* as these protocols do not rely on `diffie-hellman` builtin.
Execution: Run `sh run_trace_properties.sh`.
Results: Printed results shall be as in Table 2. These results correspond to Table 4 of our paper. To check TAMARIN results for WireGuard (with fix for anonymity based on psk), execute:

```
cd wireguard_with_fix_psk/trace_properties/tamarin
cat *.spthy.log | grep "verified\|falsified"
```

All evaluated properties shall be `verified`. To check TAMARIN results for Hybrid-WireGuard, execute:

```
cd hybrid_wireguard/trace_properties_necessary_conditions
cat *.spthy.log | grep "verified\|falsified"
```

All evaluated properties shall be `verified`. Finally, check sufficient conditions for Hybrid:

```
cd hybrid_wireguard/trace_properties_sufficient_conditions
sh run_check-pv.sh
```

All queries shall be `false` except for `uuks` and `buks` which shall be `true`.

WireGuard (with fix for anonymity based on psk)

```
bilateral: ∅
unilateral_initiator: ∅
unilateral_responder: ∅
uniqueness_initiator: ∅
uniqueness_responder: ∅
agreement_inithello: psk & (dhsisr | sic | src)
agreement_rechello: psk & (src | eic) & (dhsisr | sic | src)
agreement_confirm: psk & (sic | erc) & (dhsisr | sic | src)
secrecy_isk7: psk & (src | eic) & (dhsisr | sic | src)
secrecy_rsk7: psk & (sic | erc) & (dhsisr | sic | src)
secrecy_mul7: psk & (sic | erc) & (src | eic) & (eic | erc) & (dhsisr
       | sic | src)
secrecy_isk7pfs: psk & (sic | erc) & (src | eic) & (eic | erc) & (
      dhsisr | sic | src)
secrecy_rsk7pfs: psk & (sic | erc) & (src | eic) & (eic | erc) & (
      dhsisr | sic | src)
secrecy_mul7pfs: psk & (sic | erc) & (src | eic) & (eic | erc) & (
      dhsisr | sic | src)
```

PQ-WireGuard

```
uniqueness_initiator: ∅
bilateral: eipq | re
uniqueness_responder: ∅
unilateral_initiator: psk & (sipq | rr) & (sipq | sigr) & (eipq | re)
secrecy_isk7: psk & (srpq | ri) & (srpq | sigi)
unilateral_responder: psk & (srpq | ri) & (srpq | sigi) & (eipq | re)
agreement_inithello: psk
secrecy_rsk7: psk & (sipq | rr) & (sipq | sigr)
agreement_rechello: psk & (srpq | ri) & (srpq | sigi)
agreement_confirm: psk & (sipq | rr) & (sipq | sigr)
secrecy_isk7pfs: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (
      srpq | sigi) & (eipq | re)
secrecy_mut7: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (srpq
       | sigi) & (eipq | re)
secrecy_rsk7pfs: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (
      srpq | sigi) & (eipq | re)
secrecy_mut7pfs: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (
      srpq | sigi) & (eipq | re)
```

PQ-WireGuard⋆

```
unilateral_responder: ∅
bilateral: ∅
uniqueness_responder: ∅
uniqueness_initiator: ∅
unilateral_initiator: ∅
secrecy_isk7: psk & (srpq | ri)
agreement_confirm: psk & (sipq | rr)
agreement_rechello: psk & (srpq | ri)
secrecy_rsk7: psk & (sipq | rr)
secrecy_mut7: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
agreement_inithello: psk
secrecy_isk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
secrecy_mut7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
secrecy_rsk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
```

Hybrid-WireGuard

```
unilateral_responder: ∅
unilateral_initiator: ∅
bilateral: ∅
uniqueness_initiator: ∅
uniqueness_responder: ∅
secrecy_isk7: psk & (srpq | ri) & (src | eic) & (dhsisr | sic | src)
agreement_inithello: psk & (dhsisr | sic | src)
secrecy_rsk7: psk & (sipq | rr) & (sic | erc) & (dhsisr | sic | src)
agreement_rechello: psk & (srpq | ri) & (src | eic) & (dhsisr | sic |
       src)
agreement_confirm: psk & (sipq | rr) & (sic | erc) & (dhsisr | sic |
      src)
secrecy_mut7: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic |
      erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
secrecy_isk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic
       | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
secrecy_rsk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic
       | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
secrecy_mut7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic
       | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
```

Table 2: Results for Experiment (**E2**).

(E3): *[Anonymity evaluation] [1 human-minutes + 4 compute-hours]: evaluate all* ProVerif *and* DeepSec *files that model anonymity for WireGuard (with fix for anonymity based on* psk*), PQ-WireGuard, PQ-WireGuard⋆, Hybrid-WireGuard.*

Preparation: Inside Docker image (subsubsection A.3.2), in folder `artifacts_evaluation`, file `run_anonymity.sh` contains all necessary commands that allows to, sequentially, evaluate all files for anonymity for all protocols. These evaluations use ProVerif verifier, for all protocols, and DeepSec verifier for PQ-WireGuard and PQ-WireGuard⋆. We could not use DeepSec verifier for WireGuard (with fix for anonymity based on psk) and Hybrid-WireGuard as DeepSec does not handle exponentiation.

Execution: Run `sh run_anonymity.sh`.

Results for WireGuard (with fix for anonymity based on psk). For ProVerif, execute:

```
cd wireguard_with_fix_psk/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `initiator`, `-Eic`, `-Psk`, `-Src` and for files `responder`, `-Eic`, `-Psk`, `-Src` and shall be `Observational equivalence is true` for files `initiator`, `-no-reveal`, `-Erc-Sic` and for files `responder`, `-no-reveal`, `-Erc-Sic`.

Results for PQ-WireGuard. For DeepSec, execute:

```
cd pq_wireguard/equivalence_properties/Anonymity
sh run_check-dps.sh
```

Results shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for all files.

Results for PQ-WireGuard⋆. For DeepSec, execute:

```
cd pq_wireguard_star/equivalence_properties/Anonymity
sh run_check-dps.sh
```

Results shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard_star/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `initiator`, `-Psk`, `-Ri`, `-Rr`, `-Siq`, `-Srq` and for files `responder`, `-Psk`, `-Ri`, `-Srq` and shall be `Observational equivalence is true` for file `initiator`, `-Eiq-Re` and for file `responder`, `-Siq-Rr-Eiq-Re`.

Results for Hybrid-WireGuard. For ProVerif, execute:

```
cd hybrid_wireguard/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `initiator, -Psk, -Eic-Ri, -Eic-Srq, -Erc-Siq, -Erc-Rr, -Sic-Rr, -Sic-Siq, -Src-Ri, -Src-Srq` and for files `responder, -Psk, -Eic-Ri, -Src-Ri, -Src-Srq, -Src-Eic` and shall be `Observational equivalence is true` for files `initiator, -all-dh, -all-pq, -no-reveal, -Sic -Erc-Srq-Eiq-Ri-Re, -Sic-Src-Eic-Erc-Eiq-Re, -Src-Eic-Siq-Eiq-Rr-Re` and for files `responder, -all-dh, -all-pq, -no-reveal, -Sic -Erc-Siq-Srq-Eiq-Ri-Rr-Re, -Sic-Src-Eic-Erc -Siq-Eiq-Rr-Re`.

(E4): *[Strong secrecy evaluation] [2 human-minutes + 10 compute-hours]: evaluate all* ProVerif *and* DeepSec *files that model strong secrecy for WireGuard (with fix for anonymity based on* psk*), PQ-WireGuard, PQ-WireGuard⋆, Hybrid-WireGuard.*

Preparation: Inside Docker image (subsubsection A.3.2), in folder `artifacts_evaluation`, file `run_strong_secrecy.sh` contains all necessary commands that allows to, sequentially, perform all evaluations. These use ProVerif verifier, for all protocols, and DeepSec verifier for PQ-WireGuard and PQ-WireGuard⋆. We could not use DeepSec verifier for WireGuard (with fix for anonymity based on psk) and Hybrid-WireGuard as DeepSec does not handle exponentiation.

Execution: Run `sh run_strong_secrecy.sh`.

Results for WireGuard (with fix for anonymity based on psk). For ProVerif, execute:

```
cd wireguard_with_fix_psk/equivalence_properties/Strong-
     Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-Src, -Psk-DH-Eic, -Psk-Sic-Eic` and shall be `Observational equivalence is true` for files `-Psk-Eic-Erc, -Psk-Sic-Erc, -Sic-Src-Eic-Erc`.

Results for PQ-WireGuard. For DeepSec, execute:

```
cd pq_wireguard/equivalence_properties/Strong-Secrecy
sh run_check-dps.sh
```

Evaluation result shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-Srq, -Psk-Ri-Ti` and shall be `Observational equivalence is true` for files `-Psk-Siq-Ri-Rr-Re-Eiq-Tr, -Psk -Siq-Rr-Re-Ei-Tr-Ti, -reveal-all-but-Psk`.

Results for PQ-WireGuard⋆. For DeepSec, execute:

```
cd pq_wireguard_star/equivalence_properties/Strong-Secrecy
sh run_check-dps.sh
```

Results shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard_star/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-Ri, -Psk-Srq` and shall be `Observational equivalence is true` for files `-Psk-Siq-Eiq-Rr-Re, -Siq-Srq-Eiq -Ri-Rr-Re`.

Results for Hybrid-WireGuard. For ProVerif, execute:

```
cd hybrid_wireguard/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-DH-Eic-Ri, -Psk-DH-Eic-Srq, -Psk-Sic-Eic-Ri, -Psk-Sic -Eic-Srq, -Psk-Src-Ri, -Psk-Src-Srq` and shall be `Observational equivalence is true` for files `-all-dh, -all-pq, -no-reveal, -Sic-Src -Eic-Erc-Siq-Srq-Eiq-Ri-Rr-Re`.

A.5 Notes on Reusability

Our artifacts can be extended to benchmark and evaluate new versions (e.g. fixes) of the protocols we evaluate in our paper.

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/.