USENIX Security '25 Artifact Appendix: A Comprehensive Formal Security Analysis of OPC UA

Vincent Diemunsch
ANSSI & Université de Lorraine,
CNRS, Inria, LORIA, France

Lucca Hirschi
Université de Lorraine,
CNRS, Inria, LORIA, France

Steve Kremer
Université de Lorraine,
CNRS, Inria, LORIA, France

A Artifact Appendix

A.1 Abstract

Using the ProVerif cryptographic protocol verifier, we performed an in-depth formal security analysis of OPC UA, a standardized Industrial Control System (ICS) protocol. Our analysis uncovered attacks in several protocol configurations and established security proofs otherwise. In this artifact, we provide *(i)* the ProVerif models of the protocol and security properties, *(ii)* the scripts for reproducing the discovery of attacks and the security proofs (and the dependency graph between properties), and *(iii)* the vulnerability report used in the responsible disclosure.

A.2 Description & Requirements

Analyzing OPC UA with ProVerif for a given configuration. We provide the Python script `opcua.py` (configured by `config.py`) that instantiates our ProVerif model of OPC UA to a given configuration, and launches ProVerif on it. Table 1 depicts how configurations are defined by the presence or absence of the features and options and also specifies the corresponding script syntax.

Config. option	Admissible values	Script syntax
SecurityPol.	ECC | RSA	ECC | RSA
Mode	Enc | Sign | None	Encrypt | Sign | None
SessSec.	SSec | SNoAA	SSec | SNoAA
UserAuth.	Cert | Pwd | Ano	cert | pwd | anon
Leak	Ltk | Chk / Nok	lt_leaks | ch_leaks / no_leaks
Reopen	T | F	reopen / no_reopen
Switch	T | F	switch / no_switch

Table 1: Model configurations are defined by setting (possibly multiple) admissible values to each configuration option. "/" denotes mutually exclusive choices.

The script `opcua.py` relies on two Jinja2 template files: *(i)* `opcua-jinja.pv` that contains the protocol model and the queries corresponding to the security properties, and *(ii)* `config-jinja.pvl` that is used for configuring the model and ProVerif behavior. The script takes two arguments: the property to check and the configuration to consider. An example of a configuration is `ECC, Sign, reopen, SSec, pwd|cert, switch, no_leaks` (see Table 1). We provide details on script usage in the `README.md` file and in Appendix A.3.

Attack reconstruction. As explained in the companion report, ProVerif is able to automatically produce attack traces for all reported attacks. Providing `opcua.py` with the property and configuration for a given attack, ProVerif finds and reconstructs this attack. To ease this task, the script `reproduce_attacks.sh` automates those calls.

Exploring the lattice of configurations. As explained in the companion report, exploring all configurations by hand can be cumbersome and error-prone. We provide a lattice exploration tool `prove.py` to automatically explore the lattice of configurations and return the maximal configurations for which a given property holds.

Other artifacts. Some security properties rely on proving invariants in the first place. The dependencies between the properties and the invariants are provided in the file `dependencies.txt`. Note that the scripts `prove.py`, `prove.sh`, and `reproduce_proofs.py`, explained next, already take care of proving the conjunction of properties and invariants so that the end-user does not have to worry about those dependencies. We provide the dependencies for completeness. Finally, the vulnerability report is available in the file `vulnerabilities.pdf`.

A.2.1 Security, privacy, and ethical concerns

The attacks we found have been responsibly disclosed to the OPC Foundation, see `vulnerabilities.pdf`.

A.2.2 How to access

To retrieve the required files, `git clone` the following repository: `https://archive.softwareheritage.org/swh:1:rev:2f94c84f6125b7c07884369ba88e35a32032475b;origin=https://github.com/vdh-anssi/opc-ua_security`.

Note that this link has been updated with respect to the paper to include improved scripts and documentation for reproducibility of the results.

A.2.3 Hardware dependencies

The attacks described in the paper, can be found and reconstructed with ProVerif on a standard laptop with 16 GB of RAM.

However, ProVerif requires more memory to establish the security proofs. To be able to establish all proofs, one needs 150 GB of RAM. Fast CPUs speed up such computations, but are not mandatory. Note that ProVerif is not multithreaded.

The lattice exploration tool is computationally much more demanding but is parallelized and thus benefits from multi-core machines. We recommend a machine with at least 20 CPUs and 378 GB of RAM to run the lattice exploration tool for all properties.

A.2.4 Software dependencies

The results presented in the companion paper have been obtained with the development version of ProVerif, compiled from the sources on branch `improved_scope_lemma`. This is possible on Unix / MacOS / Windows platforms. Note that `Objective Caml` version 4.03 or higher is required. ProVerif can be installed from its sources available at https://gitlab.inria.fr/bblanche/proverif/. For using our scripts, you also need `Python` version 3.11 or higher with the template engine `Jinja2`.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The different alternatives to install an OCaml compiler are presented at https://ocaml.org/. Then, build ProVerif from its sources:

```
$ git clone https://gitlab.inria.fr/bblanche/proverif.git
$ cd proverif/proverif
$ git checkout 6a803aa13ccde13c0574912da93f029f86f63951
$ ./build
$ ./proverif --help
```

Make sure that the ProVerif executable is in your path and available from the command line.

Install `Python` version 3.11 or higher by following the instructions from https://www.python.org/downloads/. The template engine `Jinja2` (https://pypi.org/project/Jinja2/) can be installed with your preferred Python packet manager. For example using `pip3` (install with `sudo apt install python3-pip` on Unix platforms), one can install this dependency with the following command: `$ pip3 install jinja2`.

A.3.2 Basic Test

To check that the protocol model is indeed executable, one can query the reachability of events in a simple configuration. Note that reachability of an event returns `false`, whereas an unreachable event returns `true`, *i.e.*, the model is not able to execute a given event in the specified configuration. For example, executing: `$ python3 opcua.py -s -c "ECC, Sign, reopen, SSec, pwd|cert, switch, no_leaks"` will include (among others) the results:

```
- Query not event(S_open_channel(id_<xxx>,Sign,ECC,false))
is false.
- Query not event(S_open_channel(id_<xxx>,Encrypt,ECC,false))
is true.
```

as Sign mode is enabled, but not Enc (where `<xxx>` is an integer whose value may change among runs).

A.4 Evaluation workflow

A.4.1 Major Claims

We summarize here the claims of the companion paper, notably from Section 5 *"Analysis Results"*, that describes the attacks and the scope of the proofs.

(C1): *"Except for the signature oracle attack, all vulnerabilities we report were automatically discovered using ProVerif on our model."* See (E1).

(C2): *"Proofs for $\mathsf{Conf}_{\mathrm{C}}$ and $\mathsf{Conf}_{\mathrm{S}}$ are achieved with the maximal configuration for RSA; for ECC, we can prove the maximal configuration excluding None and SNoAA: (i) without Leak, as well as (ii) with Leak and either without Reopen, or without Switch. "*
"Proofs for $\mathsf{Conf}_{\mathrm{Pwd}}$ are for the maximal configuration without Leak. With Leak, we must choose between Reopen and Switch. We otherwise capture all configurations when considering in isolation SecurityPolicy and Mode." See (E2).

(C3): *"Proofs for $\mathsf{Agr}^{-}_{\mathrm{S}}$ can be obtained without Leak for Reopen $+$ Switch; and with Leak without Reopen nor Switch."*
"Regarding $\mathsf{Agr}^{-}_{\mathrm{C}}$, one can additionally prove it for RSA $+$ Leak $+$ Reopen $+$ Switch (without Enc)." See (E3).

(C4): *"We developed a Python script to efficiently explore the verification of all configurations for a given property, and extract maximal configurations for which the property holds, minimal configurations for which the property does not hold, as well as minimal configurations for which ProVerif does not terminate successfully."* See (E4).

A.4.2 Experiments

(E1): [Attack finding and reconstruction] [5 human-minutes + 30 compute-minutes + 150 MB disk]:
How to: For each attack, the `README.md` file provides detailed instructions on how to automatically find the attack and produce an attack trace in a PDF file (or, if

applicable, inspect how ProVerif constructs a clause that contradicts the query).

Execution: Run `./reproduce_attacks.sh`, which calls `opcua.py` for each of the attacks.

Alternatively: run `opcua.py` manually. For instance, one can refind the attack *§5.1 Race Condition for User Contexts* by executing `$ python3 opcua.py -q "3.1.race" -c "ECC, Encrypt, no_reopen, SSec, cert, no_switch, no_leaks" --html`.

Results: `./reproduce_attacks.sh` takes less than 30 minutes to execute and stores results in several `output_<attack>` directories (where `attack` refers to the specific attack). The complete output can be browsed in `index.html` of that folder and attack traces are available in `tracei.pdf` (where `i` is an integer, typically `1` or `2`). Specifically, the race condition attack is found and reconstructed in less than 10 s, and depicted in the file `output/trace1.pdf` when produced with `opcua.py`.

(E2): [Confidentiality properties: $\mathit{Conf}_{\mathrm{C}}$, $\mathit{Conf}_{\mathrm{S}}$ & $\mathit{Conf}_{\mathrm{Pwd}}$] [1 human-hour + 10 compute-hours]:

How to: The `results.md` file provides detailed instructions on how to launch ProVerif with the maximal configurations that back up our claim (C2).

Execution: Only the command for the first configuration for $\mathit{Conf}_{\mathrm{C}}$ is shown here. The others are similar.

```
$ python3 opcua.py -q "Conf[C]" -c "RSA,
None|Sign|Encrypt, reopen, SNoAA|SSec,
anon|pwd|cert, switch, lt_leaks"
```

Results: All commands terminate with a ProVerif verification summary stating that the query is `true`.

(E3): [Weakened agreement properties: $\mathit{Agr}^{-}_{\mathrm{S}}$ and $\mathit{Agr}^{-}_{\mathrm{C}}$] [1 human-hour + 24 compute-hours]:

How to: The `results.md` file provides detailed instructions on how to launch ProVerif with the maximal configurations that back up our claim (C3). In particular, it describes which invariants must first be proven and how.

Preparation: The property and its invariants to prove can be launched in parallel on a multi-core machine. Note however that some queries can be very demanding, for instance `3.1.B` in configuration ECC $+$ Enc $+$ Cert $+$ Leak required 23 compute-hours and 30 GB of memory.

Execution: Only the command for proving the main property $\mathit{Agr}^{-}_{\mathrm{S}}$ without its invariants (called "3.1" in our model) is shown here: `$ python3 opcua.py -q "3.1" -c "RSA, Encrypt, no_reopen, SNoAA, cert, no_switch, lt_leaks"`. All other required invariants, according to `dependencies.txt`, must be proved with the same configuration.

Because this can be cumbersome, we additionally provide a helper script `reproduce_proofs.py` that will prove the property and all its invariants automatically. For example, the following command will fully prove $\mathit{Agr}^{-}_{\mathrm{S}}$ for the provided configuration: `$ python3 reproduce_proofs.py -q "Agr-[S->C]" -c "RSA, Encrypt, no_reopen, SNoAA, cert, no_switch, lt_leaks"`.

Results: All commands must terminate with a ProVerif verification summary stating that all queries are `true`. The first command proving `3.1` takes less than 3 seconds, the command proving $\mathit{Agr}^{-}_{\mathrm{S}}$ and its invariant concludes in less than 15 seconds. Some other properties and configurations from `results.md` are more computationally demanding, hence the overall time estimation.

(E4): [Launching lattice exploration campaigns] [15 human-minutes + 24 compute-hours]:

How to: The last section of `README.md` describes in details how to use `prove.sh` to launch complete lattice exploration campaigns.

Preparation: Lattice exploration campaigns will launch many different ProVerif runs and is thus best suited for a server with many cores and a lot of memory (see Appendix A.2.3).

Execution: A campaign on the property $\mathit{Conf}_{\mathrm{C}}$ can be launched with `$ ./prove.sh "Conf[C]" "RSA, None|Sign|Encrypt, reopen, SNoAA|SSec, anon|pwd|cert, switch, lt_leaks"` (note the second argument that defines the maximal configuration the campaign aims for; here the maximal configuration without ECC). The script first asks for a starting configuration, press `Enter` to use the minimal one.

Results: The script displays: 1. "the minimal FALSE configurations" for which an attack was found, 2. "the minimal configurations" which could not be proved with the current resources budget (time and RAM), 3. "the maximal configurations" which were proven.

For the above command and after less than 20 core-minutes, the script will have ended the exploration and shows the proven maximal configuration (which was the maximal provided configuration).

Note that for other properties, some configurations may not be proven at all and the exploration will not terminate. We kill the script after some time, when we are able to extract the wanted information from the results.

A.5 Notes on Reusability

Our ProVerif models can be extended to explore new protocol evolutions and fixes. Our lattice exploration tool `prove.py` can be adapted to other protocols, see for example our generic lattice class `Trie` and `mark_trie` function.

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/`.