ACM CCS '23 Artifact Appendix: A Generic Methodology for the Modular Verification of Security Protocol Implementations

Linard Arquint
Department of Computer Science
ETH Zurich, Switzerland

Malte Schwerhoff
Department of Computer Science
ETH Zurich, Switzerland

Vaibhav Mehta
Cornell University
Ithaca, NY, USA

Peter Müller
Department of Computer Science
ETH Zurich, Switzerland

Keywords

Protocol implementation verification, Symbolic security, Separation logic, Automated verification, Injective agreement, Forward secrecy.

A Artifact Appendix

A.1 Abstract

Security protocols are essential building blocks of modern IT systems. Subtle flaws in their design or implementation may compromise the security of entire systems. It is, thus, important to prove the absence of such flaws through formal verification. Much existing work focuses on the verification of protocol *models*, which is not sufficient to show that their *implementations* are actually secure. Verification techniques for protocol implementations (e.g., via code generation or model extraction) typically impose severe restrictions on the used programming language and code design, which may lead to sub-optimal implementations. In this paper, we present a methodology for the modular verification of strong security properties directly on the level of the protocol implementations. Our methodology leverages state-of-the-art verification logics and tools to support a wide range of implementations and programming languages. We demonstrate its effectiveness by verifying memory safety and security of Go implementations of the Needham-Schroeder-Lowe, Diffie-Hellman key exchange, and WireGuard protocols, including forward secrecy and injective agreement for WireGuard. We also show that our methodology is agnostic to a particular language or program verifier with a prototype implementation for C. Hence, our work comprises a reusable verification library implementing our methodology in Go and prototypically in C, a pen-and-paper sketch of our soundness proof, and four verified implementations of security protocols: Needham-Schroeder-Lowe implemented in C and Go implementations of Needham-Schroeder-Lowe, the Diffie-Hellman key exchange, and WireGuard protocols.

A.2 Description & Requirements

This appendix describes the artifact that accompanies our paper [1] and provides all necessary information to evaluate our artifact. We provide all necessary tools and dependencies as a ready-to-use Docker image. Hence, we only require a system on which Docker is already installed. However, longer verification times than reported in the paper are to be expected because Docker adds a significant virtualization overhead compared to natively running the program verifiers.

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, and ethical concerns because the program verifiers all run in Docker environments and do not perform destructive operations.

A.2.2 How to access

All source code is available open-source on Zenodo [3] and additionally on GitHub with an appropriate tag [4] for a better browsing experience of the source code and continuous integration for our proofs. The Docker image is hosted on GitHub and provides a reproducible build and verification environment ensuring that our results are reproducible. The pen-and-paper sketch of our soundness proof is part of the paper's extended version, which we have published on arXiv [2].

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

We require an installation of Docker. The following steps have been tested on macOS 14.0 with the latest version of Docker Desktop, which is at time of writing 4.24.2 and comes with version 24.0.6 of the Docker CLI.

A.2.5 Benchmarks

In the paper, we report performance of our program verifiers verifying the verification libraries and protocol implementations. As such, no additional data besides the source code is necessary.

A.3 Set-up

A.3.1 Installation

(1) We recommend to adapt the Docker settings to provide sufficient resources to Docker. We have tested our artifact on a 2019 16-inch MacBook Pro with 2.3 GHz 8-Core Intel Core i9 running macOS Sonoma 14.0 and configured Docker to allocate up 16 cores (which includes 8 virtual cores), 6 GB of memory, and 1 GB of swap memory. In case you are using an ARM-based Mac, enable the option "Use Rosetta for x86/amd64 emulation on Apple Silicon" in the Docker Desktop Settings, which is available on macOS 13 or newer. Measurements on an Apple M1 Pro Silicon have shown that performing this additional emulation results in 20-25% longer verification times compared to those reported in the remainder of this artifact appendix.

```
docker run -it --platform linux/amd64 --volume $PWD/C-sync:/gobra/C --volume $PWD/Go-sync:/gobra/Go
    ghcr.io/viperproject/securityprotocolimplementations-artifact:latest
```

Figure 1: Docker command to download and start the artifact.

(2) Navigate to a convenient folder, in which directories can be created for the purpose of this artifact evaluation.
(3) Open a shell at this folder location.
(4) Create two new folders named `Go-sync` and `C-sync` by executing:
`mkdir Go-sync && mkdir C-sync`
(5) Download and start the Docker image containing our artifact by executing the command provided in Fig. 1. Note that this command results in the Docker container writing files to the two folders `Go-sync` and `C-sync` on your host machine. Thus, make sure that these folders are indeed empty and previous modifications that you have made to files in these folders (by following the evaluation workflow below) have been saved elsewhere!
(6) The Docker command above not only starts a Docker container and provides you with a shell within this container but it also synchronizes all files constituting our artifact with the two folders `Go-sync` and `C-sync` on your host machine. I.e., the local folders `Go-sync` and `C-sync` are synchronized with `/gobra/Go` and `/gobra/C` within the Docker container, respectively.

A.3.2 Basic Test

Our artifact uses the Go verifier called Gobra and C verifier called VeriFast. To ensure that these verifiers work as intended, please execute the following commands in the shell within the started Docker container:

(1) `/gobra/verify-nsl-alternative.sh` executes Gobra and verifies an initiator implementation in Go of the Needham-Schroeder-Lowe (NSL) protocol. This command takes about 1.5 min and should result in zero verification errors. Verification warnings can safely be ignored.
(2) `/gobra/verify-c-library.sh` executes VeriFast and verifies the reusable verification library for C. This command takes about 1 sec and should result in an output stating for eleven C files that zero errors were found for each file.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We demonstrate with several case studies in Go and C that our methodology is generic, i.e., is protocol-independent, programming language-independent, and program verifier-independent only requiring some common separation logic reasoning features. In the paper, these case studies are described in Sec. 6 and Fig. 14 provides lines of code, lines of specification, and verification times for these case studies. Experiments (E3 - E7) reproduce this figure.
(C2): Our methodology is reusable because the methodology can be implemented as a verification library that can be reused for verifying different protocol implementations. We implemented our methodology in a library for Go and as a prototype in C. In the paper, Sec. 5 describes this reusable verification library including the requirements that must be met by a program verifier. Our artifact contains the reusable verification library in Go for the Gobra verifier and the prototype in C for the VeriFast verifier. Fig. 9 reports the corresponding lines of code, lines of specification, and verification time. Experiments (E1) and (E2) reproduce the numbers stated in the paper's Fig. 9 for Gobra and VeriFast, respectively.
(C3): Our methodology is sound, i.e., verification correctly fails for implementations that do not achieve the intended security properties. The paper's extended version [2] provides a pen-and-paper sketch of our soundness proof. Additionally, experiment (E8) provides empirical evidence that verification correctly fails for faulty implementations by seeding a bug on purpose.
(C4): Our methodology handles existing implementations because we support arbitrary code structures and support various programming language features such as heap data structures and concurrency. Additionally, verification is performed directly on the level of the source language. Thus, the verified implementations can be compiled and executed without having to generate source code first, which distinguishes us from related work. Experiments (E9 - E12) compile and execute our case studies in C and Go. All case studies store application state on the heap and the initiator implementation of WireGuard uses Go routines, i.e., lightweight threads, to send and receive transport packets in parallel.

A.4.2 Experiments

(E1): *[Go reusable verification library]*
[1 human-minute + 6 compute-minutes]: Verify our reusable verification library in Go using the Gobra program verifier.
Preparation: Open a shell within the provided artifact Docker container, as explained in Sec. A.3.1.
Execution: Execute the following command within the Docker container, which invokes Gobra to verify each Go package constituting the library:
`time /gobra/verify-library.sh`
Results: Gobra reports the verification result for each verified Go package. Additionally, Gobra provides at the end a summary over all verified packages listing some verification warnings, which can be safely ignored, zero verification errors, and the information that a report containing all the details has been written to a JSON file. After Gobra's output, the `time` command provides timing information. We use the time reported as `real`, representing the elapsed wall-clock time. Note that we see significant differences between machines and whether we run the commands in the Docker container or natively. For example, this experiment takes

Case Study		Command	Est. verification time [s]
Go/Gobra			
NSL	(E3)	`time /gobra/verify-nsl.sh`	315.4
NSL (alt.)	(E4)	`time /gobra/verify-nsl-alternative.sh`	86.0
Signed DH	(E5)	`time /gobra/verify-dh.sh`	330.7
WireGuard	(E6)	`time /gobra/verify-wireguard.sh`	769.3
C/VeriFast			
NSL	(E7)	`time /gobra/verify-c-nsl.sh`	7.5

Figure 2: Commands to execute experiments (E3) - (E7) and estimated verification time when executing the command within the Docker container. Note that these verification times vary based on the resources that are made available to Docker and are noticeably higher than reported in the paper due to Docker's virtualization overhead.

a bit more than 2 min on an M1 Mac mini when executed natively versus 5.75 min on an Intel MacBook Pro within the provided Docker container.

(E2): *[C reusable verification library]*
[1 human-minute + 1 compute-minute]: Verify our prototype of the reusable verification library in C using the VeriFast program verifier.

Preparation: Open a shell within the provided artifact Docker container, as explained in Sec. A.3.1.

Execution: Invoke the VeriFast program verifier by executing the command `time /gobra/verify-c-library.sh` within the Docker container, which verifies each C file constituting the library.

Results: VeriFast reports the number of errors and the number of statements verified for each C file. VeriFast should report zero verification errors for all files. Similarly to (E1), we use the `real` time as reported by `time` and we also see an increase in verification time compared to natively running VeriFast (0.9 sec in the Docker container vs 0.8 sec natively as stated in the paper).

(E3)-(E7): *[Verify case studies]*
[5 human-minutes + 26 compute-minutes]: Verify all our case studies in Go and C. Since the instructions for verifying our Needham-Schroeder-Lowe (NSL), signed Diffie-Hellman (DH) key exchange, and WireGuard case studies in Go and NSL in C are similar, we combine their descriptions.

Preparation: Open a shell within the provided artifact Docker container, as explained in Sec. A.3.1.

Execution: Execute the command as listed in Fig. 2 within the Docker container, which invokes the Gobra or VeriFast program verifier. Experiments (E3), (E5), (E6), and (E7) include verification of the initiator and responder implementations of the corresponding protocol, protocol-specific lemmas, and trace invariant. Experiment (E4) verifies an alternative implementation of the NSL initiator implementation, which splits the implementation into several methods to demonstrate that our methodology is not sensitive to a particular code structure, as explained in Sec. 6.1 in the paper.

Results: Similarly to experiments (E1) and (E2), no verification errors should occur and we consider the `real` time as reported by `time` as verification time. These verification times can again be noticeably higher compared to those reported in the paper due to the virtualization overhead introduced by Docker.

(E8): *[Seed a verification bug]*
[5 human-minutes + 2 compute-minutes]: Introduce a bug into our DH case study, which correctly results in a verification error. We demonstrate in this experiment that our methodology successfully catches programming mistakes that would otherwise result in a violation of a security property. We do so by attempting to send a Diffie-Hellman private key (instead of the corresponding public key) to the network. Note that we limit this experiment to our DH case study to keep the efforts needed to evaluate our artifact reasonably low. However, similar bugs can be seeded in all our case studies.

Preparation: Open a shell within the provided artifact Docker container, as explained in Sec. A.3.1, and open in a text editor of your choice the following file in your local filesystem (relative to the directory chosen in step (2) in Sec. A.3.1): `Go-sync/dh/initiator/initiator.go`

Execution: Change line 142 by replacing `X`, i.e., the initiator's DH public key, to `x`, which is the corresponding DH secret key. Additionally, change line 150 by replacing `XT` with `xT`. `XT` and `xT` are the symbolic term representation of `X` and `x`, respectively. We pass these symbolic term representations to library calls such as `Send` because we reason about the attacker and security properties symbolically, i.e., assuming perfect cryptography.
Line 142 creates the protocol's first message, which is serialized on line 144 and then sent on line 150. Thus, using the DH secret key instead of the corresponding DH public key during message construction will result in a violation of the DH secret key's secrecy. Save these modifications and execute the command `time /gobra/verify-dh.sh` within the Docker container.

Results: The verification of the responder implementation and the protocol-specific lemmas and trace invariant would still succeed because we have only touched the initiator implementation. Thanks to modularity, changes to a method body only affect the verification result of the modified method. However, since verifying the initiator results in a verification error, the script is configured to exit after encountering verification errors within a package and thus the script does not invoke

```
1 Error at: </gobra/Go/dh/initiator/initiator.go:150:8>
      Precondition of call a.llib.Send(a.IdA, a.IdB,
      msg1Data, xT) might not hold.
2 Assertion tri.messageInv(l.Ctx(), idSender,
      idReceiver, msgT, l.Snapshot()) might not hold.
```

Figure 3: Expected verification error for experiment (E8), which seeds a bug in the Diffie-Hellman case study.

Gobra to verify the responder and the protocol-specific lemmas and trace invariant.
Gobra correctly reports the verification error (after about 73 s) as shown in Fig. 3 on line 150 while verifying the initiator implementation because the faulty message does not satisfy the send operation's precondition, which requires that any message satisfies the message invariant. The message invariant ensures that no secrets leak to the network.
Note that performing only one of the two described modifications to the source file also correctly results in a verification error on line 150 because the serialized message passed to `Send` does not match its symbolic term representation.

(E9)-(E12): *[Execute case studies]*
[2 human-minutes + 2 compute-minutes]: Execute our Needham-Schroeder-Lowe (NSL), signed Diffie-Hellman (DH) key exchange, and WireGuard case studies in Go and NSL in C.
Our case studies are fully executable. These experiments demonstrate how to compile and execute the case studies.

Preparation: Open a shell within the provided artifact Docker container, as explained in Sec. A.3.1. If you've just conducted experiment (E8), please remember to undo your modifications. One way to do so is to restart the Docker container, i.e., execute `exit` within the Docker container and then the command provided in Fig. 1. Note that this overwrites all changes performed to the folders `$PWD/C-sync` and `$PWD/Go-sync`.

Execution: Execute the command as listed in Fig. 4 within the Docker container, which invokes the Go or C compiler and then runs the resulting binaries.

Results: The experiments (E9) and (E12) output the two random numbers `na` and `nb` that the initiator and responder obtain by running the protocol with each other. Both should agree on the same values for `na` and `nb`. Experiment (E10) prints the shared Diffie-Hellman secret that the initiator and responder compute, which should be identical. Experiment (E11) executes WireGuard, i.e., first establishes a WireGuard VPN connection between the initiator and responder and then uses this VPN connection to send ASCII strings. In particular, the initiator sends `"Hello"`, `"World!"`, and `""` while the responder sends `"Hello back"`, `"I'm the responder"`, and `""`. Upon reception of such a message, the corresponding receiver prints the message to standard output. After receiving an empty string, the implementations exit their run loop and terminate.

Case Study		Command
Go		
NSL	(E9)	`/gobra/test-nsl.sh`
Signed DH	(E10)	`/gobra/test-dh.sh`
WireGuard	(E11)	`/gobra/test-wireguard.sh`
C		
NSL	(E12)	`/gobra/test-c-nsl.sh`

Figure 4: Commands to execute experiments (E9) - (E12) within the Docker container.

A.5 Notes on Reusability

Our work is reusable by other researchers thanks to three reasons we would like to point out: First, Sec. 5 in the paper provides several insights we gained by implementing our reusable verification library for two different programming languages and program verifiers. These insights should speed up adding support for a third programming language or program verifier because this third program verifier is likely to share some reasoning features with one of the currently support program verifiers, e.g., regarding parametricity. Second, we make our reusable verification library (in Go and C) and case studies available open-source (with the permissive Mozilla Public License (MPL) 2.0) on Zenodo and GitHub. Third, our GitHub repository features full continuous integration, i.e., compiles, verifies, and runs all reusable verification libraries and case studies. Additionally, we also build and test the Docker image representing our artifact. Hence, other researches cannot only use the same Docker image as used for artifact evaluation but they can also inspect our scripts and fully reproduce our build and verification environment. We hope this further eases adoption of our work.

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/acmccs2023/.

References

[1] L. Arquint, M. Schwerhoff, V. Mehta, and P. Müller, "A generic methodology for the modular verification of security protocol implementations," in *CCS*. ACM, 2023, pp. 1377–1391.
[2] ——, "A generic methodology for the modular verification of security protocol implementations (extended version)," 2023. [Online]. Available: https://arxiv.org/abs/2212.02626
[3] ——, "A generic methodology for the modular verification of security protocol implementations," Dec. 2023, artifact containing the reusable verification libraries and the case studies. [Online]. Available: https://doi.org/10.5281/zenodo.8330913
[4] ——, "A generic methodology for the modular verification of security protocol implementations," Dec. 2023, artifact on GitHub containing the reusable verification libraries and the case studies. [Online]. Available: https://github.com/viperproject/SecurityProtocolImplementations/tree/CCS_23