ACM CCS'23 Artifact Appendix: Assume but Verify: Deductive Verification of Leaked Information in Concurrent Applications

Toby Murray, University of Melbourne, Australia, toby.murray@unimelb.edu.au
Mukesh Tiwari, University of Cambridge, United Kingdom, mt883@cam.ac.uk
Gidon Ernst, LMU Munich, Germany, gidon.ernst@lmu.de
David A. Naumann, Stevens Institute of Technology, USA, naumann@cs.stevens.edu

1 Artifact Appendix

This artifact appendix is a self-contained document which describes a roadmap for the evaluation of the artifact for the paper *Assume but Verify: Deductive Verification of Leaked Information in Concurrent Applications* at the 30th ACM Conference on Computer and Communications Security (CCS).

1.1 Abstract

The paper presents an approach to specify and verify expressive declassification policies for systems software. The contribution encompasses the theoretical approach as well as a practical demonstration. This artifact contains the corresponding soundness proofs mechanized in Isabelle/HOL, a tool implementation, and multiple case studies.

1.2 Description & Requirements

1.2.1 Security, privacy, and ethical concerns

None. The usual disclaimer applies: This is free software that comes with no warranties whatsoever.

1.2.2 How to access

- Artifact archive: `https://dx.doi.org/10.5281/zenodo.10037769`
- Project website: `https://covern.org/`
- Tool repository: `https://bitbucket.org/covern/secc/`

1.2.3 Hardware dependencies

None.

1.2.4 Software dependencies

For validating the proofs that are part of the Isabelle/HOL formalization

- Isabelle/HOL, tested with current Isabelle2023 and with Isabelle2022 , available at `https://isabelle.in.tum.de/index.html`.

For verifying the case studies

- Java (tested with Java version 17 and 18), *and*
- Linux (tested with Ubuntu 16.04 and 18.04 as well as recent Arch Linux), *or*
- Mac OS (tested with 10.13 and 10.14)

For running the case studies

- Bash
- A C compiler, e.g. `gcc` or `clang`, invoked as binary `cc` on `PATH` (can be overridden)

1.2.5 Case Studies (was: Benchmarks)

The contribution of this paper encompasses four case studies. Details are described in the paper, resp. Readme files and comments in the source code.

- A differentially private location server service, located at `examples/case-studies/location-server.c`
- A sealed bid auction server, located at `examples/case-studies/auction.c`
- A verified implementation of the game Wordle, located at `examples/case-studies/wordle.c`.

  Additional background on this case study can be found in several blog posts by the first author: `https://verse.systems/blog/post/`
- A private learning application, located at `examples/case-studies/private-learning.c`

1.3 Set-up

1.3.1 Installation

We recommend to install all dependencies via the package manager of your system/distribution.

Manual installation

- Download and installation of Isabelle/HOL is described at https://isabelle.in.tum.de/installation.html for multiple platforms.

  For Linux:

  ```
  wget https://isabelle.in.tum.de/dist/ \
       Isabelle2023_linux.tar.gz
  tar -xzf Isabelle2023_linux.tar.gz
  ```

  Past versions can be found here: https://isabelle.in.tum.de/download_past.html.

- Ensure that the `isabelle` binary can be found via `PATH`. On Linux, this can be achieved e.g., by

  ```
  export PATH="$PWD/Isabelle2023/bin:$PATH"
  ```

- Java is available for manual download and installation here: https://www.java.com/de/download/manual.jsp. Since it is a very common software package, we do not provide any further installation instructions in this document.

1.3.2 Basic Test

Checking that Java is installed and executable

```
java -version
```

The output should look something like this

```
openjdk version "17.0.9" 2023-10-17
OpenJDK Runtime Environment (build 17.0.9+8)
OpenJDK 64-Bit Server VM (build 17.0.9+8, mixed mode)
```

Checking that Isabelle/HOL is installed and executable:

```
isabelle version
Isabelle2023 # expected output
```

Note: running Isabelle/HOL for the first time after installation will automatically compile various of its libraries. This may take significant amount of time (e.g up to an hour), depending on the hardware used. This initial setup is not part of the evaluation of this artifact.

Since neither step in the evaluation of this artifact takes a long time, no separate functionality test have been included.

1.4 Evaluation workflow

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] This section should include all the operational steps and experiments which must be performed to evaluate if your your artifact is functional and to validate your paper's key results and claims. For that purpose, we ask you to use the two following subsections and cross-reference the items therein as explained next.

1.4.1 Major Claims

The major claims of the paper supported by this artifact are listed below. Soundness of the approach rests on adequacy of mechanizing the definitions, as well as validity of Theorems 5.6 (the policy agnostic security guarantee) and Theorem 6.5 (the policy-specific security guarantee). The artifact contains *two independent* formalizations of the theory, one that builds on the existing one for SecCSL, which covers Theorem 5.6, and a simplified variant builds on a standard semantics of sequential programs (lacking separation logic and concurrency, which covers the entire theory including both theorems 5.6 and 6.5. While porting the second result to the SecCSL formalization would be feasible, in our opinion the significant additional effort does not appear to be worthwhile.

Concrete laims regarding soundness:

(C1): The proof rules presented in Sec 4.2 ensure Theorem 5.6 (policy-agnostic guarantee) with respect to the semantics in Sec 4.3 and the formalization of attacker knowledge in Sec 5.

(C2): The extended proof rules for audit triples in Sec 6 ensure Theorem 6.5 (policy-specific guarantee) with respect to the definitions of policy audit the corresponding release policy.

Moreover, all case studies satisfy the goals outlined in Sec 2:

(C3): All four case studies leak information only via failed assumptions associated with `_(assume ...)` annotations placed in the program's source code, and that these information leak are bounded by the respective policies.

1.4.2 Experiments

Running the Isabelle/HOL proofs:

(E1): [Soundness of SecCSL + Theorem 5.6] Expected effort: up to five compute-minutes (+ 10min up to 1h for the initial Isabelle/HOL library compilation).
This experiment validates **(C1)**.
Preparation: Switch to folder `seccsl_isabelle` in the artifact.
Execution: `isabelle build -c -d . -v SecCSL`
Results: The expected output looks like this:

```
[...]

Session Pure/Pure
```

```
Session Misc/Tools
Session HOL/HOL (main)
Session Unsorted/SecCSL
Cleaned SecCSL
Running SecCSL ...
SecCSL: theory SecCSL.Syntax
SecCSL: theory SecCSL.Semantics
SecCSL: theory SecCSL.Separation
SecCSL: theory SecCSL.Logic
SecCSL: theory SecCSL.Locks
SecCSL: theory SecCSL.Shared
SecCSL: theory SecCSL.Soundness
SecCSL: theory SecCSL.SecCSL
SecCSL: theory SecCSL.Knowledge
Timing SecCSL ([...])
Finished SecCSL (0:01:13 elapsed time [...])
```

A failed proof would appear in the output, e.g., lines like this together with diagnostics information, but this should not happen.

```
SecCSL FAILED ([...])
```

(E2): [Soundness of the approach: Theorems 5.6 and 6.5] Expected effort: around 1 compute-minute.
This experiment validates **(C1)** as well as **(C2)**.
Preparation: Switch to folder `audit_isabelle` in the artifact.
Execution: `isabelle build -c -d . -v Audit`
Results: The expected output looks like this:

```
[...]

Session Pure/Pure
Session Misc/Tools
Session HOL/HOL (main)
Session Unsorted/SecCSL
Cleaned Audit
Running Audit ...
Audit: theory Audit.Commands
Audit: theory Audit.Secure
Audit: theory Audit.Guarantee
Audit: theory Audit.Rules
Audit: theory Audit.Policy
Timing Audit ([...])
Finished Audit (0:00:23 elapsed time [...])
```

(E3): [Verify the case-studies] Expected effort: 5 human minutes and 1 compute minute.
Preparation: Switch to folder `verdeca-tool` in the artifact.
This experiment validates **(C3)**.
Compile Verdeca (takes a few minutes and may need internet access to download dependencies):

```
make
```

which should ultimately print

```
[...]
[info] done compiling
[37/37] secc.jar
out/secc/launcher.dest/run
./mill secc.launcher
[41/41] secc.launcher
[echo]  Verdeca.sh
[chmod] Verdeca.sh
```

Verify the example from the motivation as follows:

```
./Verdeca.sh examples/case-studies/average.c
```

The expected output is the name of the file, followed by the status of correctness of each of the functions contained withing, for example:

```
examples/case-studies/average.c
  avg_sum_thread ...    success ♥ (time 322ms)
  avg_declass_thread ...    success ♥ (time 94ms)
```

Execution: Verify the case studies with the following commands:

```
./Verdeca.sh examples/case-studies/location-server.c
./Verdeca.sh examples/case-studies/auction.c
./Verdeca.sh examples/case-studies/wordle.c
./Verdeca.sh examples/case-studies/private-learning.c
```

Results: For each function, Verdeca should print `success` ♥. A function that fails to verify will lead to a long symbolic trace to be printed with a clear indication of the proof obligation that failed at the top.

1.4.3 Theories and Tool

Please refer to the individual `README.md` files in the subfolders to find the respective definitions and theorems in the Isabelle sources, and for further information on how to use Verdeca.

1.5 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/.