Verifying the Security of a PGP Keyserver (Abstract for the VerifyThis challenge 2020)

Gidon Ernst[1], Toby Murray[2], and Mukesh Tiwari[2]

[1] LMU Munich, gidon.ernst@lmu.de
[2] University of Melbourne, Australia, firstname.lastname@unimelb.edu.au

1 Introduction

We discuss our approach and progress concerning the formal verification of the HAGRID PGP keyserver, as part of the VerifyThis 2020 Collaborative Long-term Verification Challenge.

2 Approach

We have followed an iterative approach to the formalisation of the case study and its verification, building on the work of related teams [3]. Specifically, we took Ernst and Rieger's Scala model of the key server as a starting point. This model represents the state of the key server as a collection of maps, and has a small number of top-level functions that manipulate these maps and simulate actions like sending emails.

2.1 Abstract Specification

We began by constructing an abstract specification for the Scala model, in the spirit of typical Alloy specifications [4]. Here, the state is modelled as a collection of partial functions, each of which represents a map in the Scala model. For instance, the *keys* map stores the uploaded keys, indexed by their fingerprints; the *uploaded* map remembers which keys have been uploaded and is indexed by the token that was issued for each; the *prev-tokens* set remembers previously issued tokens (in order to specify that newly generated tokens should be fresh).

record *state* =
  keys :: *fingerprint* $\rightharpoonup$ *key*
  uploaded :: *token* $\rightharpoonup$ *fingerprint*
  prev-tokens :: *token* set
  ...

Top-level operations of the Scala model are then specified as relations on these states, describing how the state after the operation is related to the state before the operation. These specifications are carefully written to delineate pre-conditions and postconditions.

For instance, the precondition for the *upload* operation to upload a key k is a predicate on the pre-state s, and states that if a key with the same fingerprint of k has already been uploaded, then that key must be identical to k:

$$\textit{upload-pre}(k,s) \equiv k.\textit{fingerprint} \in \textit{dom}(s.\textit{keys}) \implies s.\textit{keys}(k.\textit{fingerprint}) = k$$

The postcondition for the *upload* operation requires that a fresh token is generated for the key and that the key is added to the *keys* and *uploaded* maps.

$$\begin{aligned}\textit{upload-post}(k,s,s') \equiv \exists\ t.\ & t \notin s.\textit{prev-tokens} \wedge \\ & s'.\textit{keys} = s.\textit{keys} \cup (k.\textit{fingerprint} \mapsto k) \wedge \\ & s'.\textit{uploaded} = s.\textit{uploaded} \cup (t \mapsto k.\textit{fingerprint}) \wedge \\ & s'.\textit{prev-tokens} = s.\textit{prev-tokens} \cup \{t\}\end{aligned}$$

Having delineated their pre- and post-conditions, operations are specified straightforwardly. For instance, the specification of the *upload* operation for uploading a key k, given pre- and post-states s and s' respectively, is:

$$\textit{upload}(k,s,s') \equiv \textbf{if}\ \textit{upload-pre}(k,s)\ \textbf{then}\ \textit{upload-post}(k,s,s')\ \textbf{else}\ s' = s$$

We have experimented with encoding this abstract specification in both Coq and Isabelle/HOL, and with formalising and proving invariant preservation over its operations.

2.2 Verified Model

While aiding clarity, the purpose of delineating pre- and post-conditions in the abstract specification was to serve as a guide for subsequent Hoare logic reasoning about the system. Specifically, while currently still in progress, the next step of our approach involves constructing a model of the system that refines the abstract specification and about which we can reason using a Hoare logic style program verifier.

In our approach, we chose to use the prototype SecC verifier, which automates program reasoning in the Hoare style logic Security Concurrent Separation Logic (SecCSL) [2]. SecCSL is a variant of Concurrent Separation Logic that allows reasoning about expressive information flow security policies, in addition to ordinary Hoare logic reasoning.

Information Flow Security Policies Such policies abound in the key server. For instance, when returning results pertaining to the lookup of a key k, data about all other keys should be considered private and not revealed. Additionally, however, whether a particular identity $id \in k.\textit{ids}$, attached to the key k should be revealed depends on whether that identity has been confirmed. Thus only confirmed identities for the key k should be considered public. The resulting information flow policy for this seemingly simple operation is therefore highly state-dependent. SecCSL and SecC provide natural support for verifying such stateful policies.

Dynamic Declassification Note, however, that the security policy is not fixed: the action of confirming an identity associated with key k effectively *declassifies* [5] the resulting identity, making it public with respect to a lookup for key k.

SecC supports this style of reasoning via **assume** statements. Whereas traditional Hoare logic verifiers are able to assume only functional properties, in SecCSL such assumptions can naturally encompass security assertions. For instance the statement "**assume** (e::`low`)" literally means "let us assume that the data contained in the expression e is known to the attacker" [1].

Ensuring Correct Declassification Assume statements are therefore powerful for reasoning about dynamic declassification policies. However, we also need to make sure that they are not mis-used. For example, it is appropriate to place an **assume** statement at the point that an identity *id* is confirmed for key k, which declassifies *id* with respect to lookups of key k. However, the **assume** statement would not be appropriate if it declassified *id* with respect to some other key *k'* or if it did so before *id* was confirmed. Therefore, how can we make sure that **assume** statements are used correctly to encode the desired declassification policy?

To address this issue, our methodology encodes the declassification policy via an extensional predicate that talks just about the program's inputs and outputs (i.e. not about its internal state), inspired by [6]. Then at the site of each **assume** statement, we immediately precede the **assume** statement with an **assert** statement to check that the extensional declassification predicate holds (i.e. allows the declassification encoded in the **assume** statement to occur).

Doing so allows us to verify expressive declassification policies naturally via **assume** statements, free of the risk that we inadvertently verify the model against the wrong policy.

References

1. Chudnov, A., Naumann, D.A.: Assuming you know: Epistemic semantics of relational annotations for expressive flow policies. In: IEEE Computer Security Foundations Symposium (CSF). pp. 189–203. IEEE (2018)
2. Ernst, G., Murray, T.: SecCSL: Security Concurrent Separation Logic. In: International Conference on Computer Aided Verification (CAV). pp. 208–230. Springer (2019)
3. Ernst, G., Rieger, L.: Information flow testing of a PGP keyserver (abstract for the verifythis challenge 2020) (2020)
4. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
5. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. Journal of Computer Security **17**(5), 517–548 (2009)
6. Schoepe, D., Murray, T., Sabelfeld, A.: Veronica: Expressive and precise concurrent information flow security. In: IEEE Computer Security Foundations Symposium (CSF) (2020), to appear.