VerifyThis
Long-term Challenge 2020

— Proceedings —

Submitted and Presented Solutions
(Online Presentation, 27 April 2020)

Edited by

Marieke Huisman
Raúl E. Monti
Mattias Ulbrich
Alexander Weigl

June 2020

Preface

The VerifyThis verification competition is a regular event run as an on-site meeting with workshop character in which three challenges are proposed that participants have to verify in 90 minutes each using their favourite program verification tool.

We have experienced that the state of the art of program verification allows the participants to specify and verify impressively complex algorithms in this short a time span. If such sophisticated, realistic but not real, problems can be solved in real-time, what would be achievable if (a) we as the program verification community collaborated and (b) the time constraints were removed?

The *VerifyThis Long-term Challenge* aims at proving that deductive program verification can produce relevant results for real systems with acceptable effort. This challenge is not designed as a competitive event. It would be nice if one group could prove the protocol of the system to be correct whereas another group would show that an implementation follows that protocol, and is hence correct. Participants could choose aspects and parts of the challenge that they wanted to specify and verify with the tool of their choice.

We chose the PGP-keyserver infrastructure called HAGRID as the system to considered in the challenge as it (a) is security- and safety-relevant, (b) has a modular system design that allows for reimplementation of chosen components in different languages and (c) raises a variety of interesting formally analysable questions.

The challenge started in August 2019, and verification reports could be submitted for presentation until the end of February 2020. After their (online) presentation and following the discussion, the community decided that the verification goals still deserve more attention, and that work should continue.

The challenge manual that was a available as a description of the challenge is included in these proceedings on page 17.

At the end of the submission period, we had received five contributions from different teams using different verification approaches and tools. Three of them focused on functional specification, two considered security-related aspects.

Submissions for functional properties were written in different programming (and specification) languages and verified with different deductive verification tools (SPARK, Why3, and KeY). All specifications followed similar ideas: They specified the functional key server interface using contracts that formally capture the effects of requests on the database (represented by some form of ADT). The Why3 solution targetted a file-based database backend, whereas the other two solutions modelled it as an in-memory database.

Two submissions provided a security testing framework for the keyserver based on history traces (in Scala), and a formulation of information flow security properties with declassification in a variant of separation logic for security properties. In particular, revocation of key entries was identified here as an interesting challenge for non-interference approaches.

A final workshop session for the challenge had been planned at ETAPS (along with the VerifyThis program verification competition). Since ETAPS has been postponed, we met and exchanged online. During the online meeting, the different solutions were briefly explained, and we discussed the approaches, how they can benefit from one another and how further verification success can be stipulated.

We would like to thank all participants of the challenge and the online-event.

June 2020

Marieke Huisman
Raúl E. Monti
Mattias Ulbrich
Alexander Weigl

Organization

Program Committee

Marieke Huisman	University of Twente, Netherlands
Raúl E. Monti	University of Twente, Netherlands
Mattias Ulbrich	Karlsruhe Institute of Technology, Germany
Alexander Weigl	Karlsruhe Institute of Technology, Germany

Submission Authors

Alexander Weigl
Claire Dross
Claude Marché
Cláudio Lourenço
Diego Diverio
Gidon Ernst
Johannes Kanig
Lukas Rieger
Mattias Ulbrich
Mukesh Tiwari
Stijn de Gouw
Toby Murray
Yannick Moy

Table of Contents

A Solution to the Long-Term Challenge in SPARK

Claire Dross, Johannes Kanig, and Yannick Moy

AdaCore, 75009 Paris

1 Introduction

SPARK [2] is a programming language designed to be amenable to formal verification. In addition to the standard features of a procedural programming language, it contains standard annotations such as pre- and postconditions and annotations of effects on global variables. SPARK is a subset of the Ada language [1], so it can be compiled to machine code, e.g. using the GNAT compiler. It can also be freely mixed with (unverified) Ada code.

We have applied SPARK to the key server described in the long-term challenge. Our goal was to get a working prototype, so we did some simplifications in order to move quicker. For example, our prototype uses in-memory storage (simple lists) to store the key server data, as opposed to using external storage. This means that if our key-server is switched off, it will lose all data.

The code is available at https://github.com/AdaCore/Lumos_Maxima.

2 Overview of the Code

The central piece of verified code is the implementation of the server API in `server.adb`. In this code we verified absence of runtime errors, but also quite strong postconditions. Here are the declarations for `Request_Add` and `Verify_Add`, including their contracts:

```
procedure Request_Add
  (Email : Email_Id;
   Key   : Key_Id;
   Token : out Token_Type)
with
  Pre  => Invariant,
  Post => Invariant
    and Contains (Seen_Tokens, Token)
    and Email = Get_Email (Seen_Tokens, Token)
    and Key = Get_Key (Seen_Tokens, Token)
    and Is_Add (Seen_Tokens, Token)
    and Seen_Tokens'Old ≤ Seen_Tokens;

procedure Verify_Add
  (Token  : Token_Type;
   Status : out Boolean)
with
  Pre  => Invariant,
  Post => Invariant
    and (if Status
         then Contains (Seen_Tokens, Token)
           and Is_Add (Seen_Tokens, Token)
           and Model'Old ≤ Model
           and Included_Except
             (Model, Model'Old,
               (Get_Key (Seen_Tokens, Token), Get_Email (Seen_Tokens, Token)))
           and Contains
             (Model,
               (Get_Key (Seen_Tokens, Token), Get_Email (Seen_Tokens, Token)))
         else Model = Model'Old)
    and Seen_Tokens'Old ≤ Seen_Tokens;
```

As one can see, we prove that `Request_Add` returns a token with the proper information attached, and this token represents a request to add a key (expressed with `Is_Add`). Once `Verify_Add` is called, the key/email pair is added to the database (this is expressed using the `Model` variable). The code for `Request_Remove` and `Verify_Remove` is similar.

In total, five units (files) with 500 lines of code have been verified with SPARK Pro 20.1, out of 10 units with 1300 lines. The total running time of the proof tool on an 8-core/16-threads AMD Ryzen 1700x is roughly 1 minute from scratch, and roughly 30 seconds to replay all proofs using an existing session.

3 Difficulties and Workarounds

We found that SPARK proofs worked best when storing basic data such as integers in the central database, and not more complex data such as strings. The reason seems to be that the containers that we use rely heavily on equality for their specifications, and the equality for complex types is much heavier and degrades proof performance. This issue prevented us from writing natural code where e.g. email addresses would be represented directly by strings. We worked around this situation by interning strings in a separate table, and referring to the table indices instead of the actual strings. The string interning code is also proved.

4 The Missions of the Challenge

We addressed mission 0 (identify relevant properties) by defining the design of our software and writing postconditions, mission 1 (safety) by proving absence of runtime errors, mission 2 (functionality) by proving the specified postconditions, and finally mission 6 (termination) by adding termination annotations. Proof of termination only required a single loop variant.

We did not address mission 3 (protocol), mission 4 (privacy), mission 5 (thread safety) and mission 7 (randomness), which are out of scope for SPARK. Proof of thread-safety is possible in SPARK, though the analysis is quite restrictive - threads cannot access memory that is potentially written by other threads unless these accesses are protected against race conditions. Also, this analysis requires the use of Ada tasking and no other mechanisms, and visibility over all tasks (not the case in our prototype, where tasks might be created by the web server).

5 Additional Unverified Code

To get a working example, we wrote some unverified Ada code. This code starts a web server using the AWS (Ada Web Server [1]) framework and dispatches incoming requests to the various API functions. It also extracts emails from the submitted public key, so that the API functions can be called directly with interned email addresses. With reasonable additional effort, the percentage of verified code could be increased, to leave only the code directly related to AWS requests unproved.

When a user connects to the interface, a welcome page is presented with links to the subpages to query existing keys by email or adding a new key. On the page to add a new key, the user can insert a public key (the begin and end markers of a typical public key need to be removed here). From this key, the code extracts the email address, creates the request token from the interned email/key pair and returns the token to the user. To simplify the setup, we show the confirmation link directly in the browser instead of emailing it. This would of course have to change for a production implementation. The code invoked by this link then inserts the (interned) email/key pair into the database.

References

1. Barnes, J.: Programming in Ada 2012. Cambridge University Press (2014)
2. McCormick, J. W., Chapin, P. C.: Building high integrity applications with SPARK. Cambridge University Press (2015)

[1] https://www.adacore.com/gnatpro/toolsuite/ada-web-server

"You-Know-Why": an Early-Stage Prototype of a Key Server Developed using Why3

Diego Diverio[1], Cláudio Lourenço[2], and Claude Marché[2]

[1] Université Paris-Saclay, Inria, 91120 Palaiseau, France
[2] Université Paris-Saclay, Univ. Paris-Sud, CNRS, Inria, LRI, 91405 Orsay, France

Abstract. This is a preliminary report on a solution we designed for the VerifyThis Collaborative Long Term Challenge 2019 [6].

1 Used verification approach and tools

We used the Why3 verification framework [1] to design, from scratch, a simple but running prototype implementation of a PGP key server. The current version does not have a Web interface but a basic command-line interface. We exploit the ability of Why3 to extract OCaml code from verified WhyML code (getting rid of formal specifications and *ghost* code [3]) so as to produce code that can be compiled into an executable.

Our prototype is made of a combination of unverified handwritten OCaml code and of WhyML code whose functional behaviour is formally specified and proven correct. As Why3 is a framework for sequential programs only, our approach does not address any issue related to concurrency of execution of server requests.

2 How the challenge was adapted to make verification possible

The main approach we took was to follow the document presenting the challenge [5]. We also had a look at the implementation of the Hagrid key server [4] written in Rust, to get more precise ideas on its implementation. In particular, we discovered that it was using the file system of the host computer as a database, instead of any more sophisticated database management system.

From the challenge document [5] we addressed mission 1 (Safety), mission 2 (Functionality) and mission 6 (Termination). The mission 0 (Identify Relevant Properties) was not deeply investigated: we turned the informal properties of the five operations given in the challenge documentation [5, Section 5] into formal specifications in WhyML. For that we reused the theories for sets and maps from the Why3 standard library. Yet it appeared that such theories were in fact under reorganisation and we are using the new versions, so that replaying our proofs today requires to install the Why3 version in the master branch of development[3].

[3] Or presumably the upcoming next release 1.3.0.

We also used character strings, which is a newly available theory in the Why3 standard library.

We did not really address mission 3 (Protocol), though our functional specifications implicitly express properties about the server protocol, such as the fact that to confirm a key addition, only a token previously issued by the ADD operation can be accepted. Missions 4 (Privacy), 5 (Thread Safety) and 7 (Randomness) are not addressed at all.

We also did not implement a Web front-end though we could have done it with unverified OCaml code.

3 What has been achieved

Our implementation is divided into three parts as it is described in the challenge presentation [5]: a front-end (here a basic CLI program), a back-end (which actually implements the requirements) and a database (here a file-based implementation of dictionaries). The development is publicly available from the GITLAB repository `https://gitlab.inria.fr/why3/verifythis2020`. The `README` file there explains how to replay the proofs, how to compile an executable and how to run the latter.

The back-end interface is specified by a WhyML module `Spec` containing an abstract view of the server global *state* - under the form of a record type with ghost fields and invariants - and the expected five operations QUERY, ADD, CONFIRMADD, DEL and CONFIRMDEL. The back-end is implemented in another module `Impl` which refines `Spec`, the type *state* being extended with concrete fields. Gluing invariants relate those concrete fields with the ghost fields. The refinement is proved correct. The proofs are not difficult: they proceed smoothly using automated provers only.

The module `Impl` makes use of an extra module `Table` providing an interface to a general data structure of dictionaries mapping strings to strings. The `Table` module is only a WhyML interface, its implementation is written in OCaml and is not verified. Yet, this OCaml code makes use of an implementation of `Base64` encoding-decoding that is used to turn arbitrary strings into valid filenames. This `Base64` module is a completely independent library that was recently verified with Why3. This constitutes the database part of our prototype.

While the back-end operations rely on preconditions to ensure properties on inputs, the front-end adds a layer which checks inputs and raises exceptions when they do not conform to preconditions. The front-end is written in WhyML but is actually completed with OCaml code that provides the simple CLI interface. The front-end also implements an initialisation function that performs sanity checks on database when starting the server (see details below).

4 Successes and challenges

The main success is that we could actually write the specifications down in WhyML, develop WhyML code that is fully proven conforming to the specifications, and also extract an executable program from it that we can interface with OCaml external code, to effectively run a basic prototype server.

The proofs were not particularly challenging. Each verification condition was most of the time discharged by one of the automated theorem provers available in Why3 (one of Alt-Ergo, CVC4, or Z3 for this proof). Adding a few extra assertions in the code was enough in the remaining cases but one. This extra case has to be handled using a few manual Why3 transformations to be discharged [2], but was quite easy to prove anyway. The most challenging part was in fact the initialisation function: the code of this function must indeed do a lot of checks on the data read from disc, so as to ensure that the resulting server state satisfies the invariant. It is also the only part of the code using loops, thus requiring loop invariants to be added.

Of course, the remaining challenges are numerous, starting from the fact that we did not address at all the issues related to concurrency, privacy, or randomness. Another challenge would be to get closer to the current Hagrid server which uses clever techniques to store less information on disc. For example, it encodes all the needed information in the confirmation tokens: Hagrid does not have to store any information about the token, as it can recover it from the token itself, whereas we have to store locally for each token which email and key it was associated with.

Another challenge in a more general point of view is the amount of time, or human effort, to achieve such a verified server: we had quite little human resources to dedicate to this case study, and we had to aim at a very simple implementation if we wanted to have a working basic prototype. Designing a verified alternative to Hagrid, with all required guarantees regarding privacy and concurrency, should start by planning significant allocation of human resources.

Acknowledgements

For their contributions and feedback, we would like to thank François Bobot, Sylvain Dailler and Jean-Christophe Filliâtre.

References

1. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer (STTT) **17**(6), 709–727 (2015). https://doi.org/10.1007/s10009-014-0314-5, see also `http://toccata.lri.fr/gallery/fm2012comp.en.html`
2. Dailler, S., Marché, C., Moy, Y.: Lightweight interactive proving inside an automatic program verifier. In: Proceedings of the Fourth Workshop on Formal Integrated Development Environment, F-IDE, Oxford, UK, July 14, 2018 (2018)
3. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods in System Design **48**(3), 152–174 (2016). https://doi.org/10.1007/s10703-016-0243-x
4. Hagrid, a verifying OpenPGP key server. Web page `https://gitlab.com/hagrid-keyserver/hagrid`

5. Huisman, M., Monti, R., Ulbrich, M., Weigl, A.: VerifyThis collaborative long term challenge: The PGP key server — challenge manual. `https://verifythis.github.io/VerifyThisLongTerm.pdf` (Aug 2019)
6. VerifyThis collaborative long term challenge. Web page `https://verifythis.github.io/challenge/` (2019)

The KeY Approach on Hagrid

VerifyThis Long-Term Challenge 2020

Stijn de Gouw, Mattias Ulbrich, and Alexander Weigl

[1] Open University
[2] Karlsruhe Institute of Technology

1 Introduction

We present the results of the application of the KeY verification approach to the VerifyThis Long Term Challenge 2020.

KeY [1] is a deductive program verification engine to show the conformance of Java Programs to their specification in the Java Modeling Language (JML). It supports sequential Java 1.4 and the full JavaCard 3.0 standard. The deductive engine of KeY is based on a sequent calculus for a dynamic logic for Java and supports both interactive and automatised verification.

2 Verification of the Subject

The verification target of the challenge is the HAGRID key server, a new implementation of the PGP key server written in Rust that makes the key server conform to data protection regulations and increases resilience against denial of service attacks.

Since KeY operates on programs written in Java, it cannot directly be used to verify HAGRID's Rust source code. Hence, a simplified re-implementation of the of the core functionality of the HAGRID key server in Java had to be written. We came up with two different Java implementations of different complexity. Both adhere to the natural language specifications in [2] The first version implements a single class that only makes use of primitive data types and arrays. The second version modularizes the first version and uses an implementation of a map data structure. Both versions are abstractions of HAGRID's implementation and actual behaviour. We focus on the database logic and leave network connection, and en- and de-coding of HTTP messages aside for this project. Moreover, in the implementation, we assume that e-mail addresses and keys are "atomic" in the sense that they are used as keys and values in the database, but are never analysed for their contents. In particular, we avoid the use of objects for the data and represent them by primitive integer values. This is of course a severe simplification, but since strings are objects in the Java programming language, they produce significantly more difficult verification conditions due to additional heap framing conditions which need to be shown.

We were able to specify and verify both implementations successfully.

Table 1. Verification in numbers of lines of code, lines of specification, applied rules, interactions, and proof obligations.

Version	lines of code	lines of spec	rule applications	interactive rule appl.	proof obligations
Plain	69	82	30.119	0	10
Map-based	146	262	77.663	89	40

A simple email-key map. The first version bases upon five integer arrays. These arrays store:

- the email (identification) of the user
- one array for confirmed and one array for unconfirmed keys
- an array that stores confirmation codes, and
- an array that stores which operation was most recently requested.

The maximum number of users is fixed to 1024, as the arrays are never resized. The implementation only allows to confirm last requested action, e.g. if a deletion is requested, a pending addition is abandoned. We avoid the use of any objects to avoid dealing with a changes of the heap, resulting in a version that is verifiable without interactions in KeY. Table 1 shows the aggregated metrics of the proofs.

We also attempted to add a 'time-out' mechanism, to cover the following aspect of the challenge:

> If the provided code *is one recently issued*, then the corresponding operation (addition/removal) is finalised.

This is easy to add in the implementation: first store the time that the user requests the operation (in an additional array), and when confirming, only approve the operation if that time was sufficiently recent. But it is problematic for specification and verification: the time limit may not yet have elapsed when the precondition (i.e. the specification) is evaluated, but it may have when the JVM determines the current time in the `confirm` method body. So we dropped the time-out aspect.

The map-based approach. The second version follows the same design principles as the first one, but aims to achieve a more object-oriented, modular architecture. To this end, the key server now contains four map data structures for the stored keys, pending additions and pending deletions.

Fig. 1 gives an overview of the class layout. The interface `KIMap` (Key Integer Map) represents a map of from `int` to `int`. Its functionality is specified (by JML contracts) using the abstract map theory built into KeY. `KIMapImpl` is a simple implementation based upon two *int*-arrays (one for the keys and one for the values). `KeyServer` is the verifying key server providing the functionality to

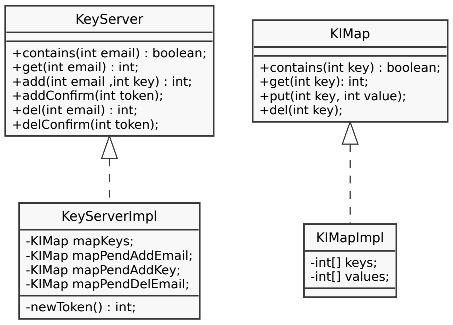

Fig. 1. UML class diagram of the Map version

answer queries for keys, to process requests for key addition and deletion and to perform these mutation operations upon confirmation. It is specified using a number of model fields containing (finite) logical maps.

3 Verification Results

We were able to verify strong functional method contracts for all methods of the implementations. This includes verifying that requested operations are confirmed by the right confirmation code, absence of runtime exceptions and a guaranteed termination of each request handler.

We noticed during the verification of the modular map approach that discharging the framing conditions brought the KeY on the edge of its capabilities. In the following, we devised a new technique to deal with framing conditions that combines dynamic frames with aspects from ownership. This allowed us to close all proofs successfully.

Both implementations used integers instead of Strings as an simplifying abstraction. The obvious next goal is the verification of an implementation which uses `String` values for e-mails and keys.

References

1. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book: From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, December 2016.
2. Marieke Huisman, Raúl Monti, Mattias Ulbrich, and Alexander Weigl. VerifyThis collaborative long term challenge: The PGP key server. `https://verifythis.github.io/VerifyThisLongTerm.pdf`, 2019. [Online; accessed 2020-02-27].

Information Flow Testing of a PGP Keyserver (Abstract for the VerifyThis challenge 2020)

Gidon Ernst and Lukas Rieger

LMU Munich, `gidon.ernst@lmu.de`

1 Abstract

We report on the progress in the VerifyThis long-term challenge 2020[1] that targets a formal analysis of the secure PGP keyserver HAGRID.[2]

A major concern in the design of the PGP keyserver HAGRID is that confirmed identities in published keys are authenticated and confidential otherwise. The underlying security property requires reasoning about *non-interference* [7], *value-dependent* information flow [10,8], as well as *declassification* [2]. We present a high-level reference model, written in Scala, that is detailed enough to capture these key concerns, but abstracts away from the internals of the server. Functional correctness and information flow security of the model is currently being analyzed using automated random testing via ScalaCheck, a variant of QuickCheck [6]. A comparison of behavior against the implementation in HAGRID is planned for future work.

2 Models

We have modeled the server, the client, the communication, and the attacker.[3] Scala is a programming language that supports both functional and object-oriented concepts, so that adequate choices regarding the abstraction level of data types and the encoding of state transitions can be made. The added benefit is that models can be executed and debugged interactively within an IDE.

Data Model: In contrast to other preliminary work in the challenge,[4] we model keys explicitly as an immutable data type that stores a set of abstract identities (i.e., email addresses) [3, Sec 5.11]. This aspect is relevant, because regardless of which identities were originally uploaded with a given key, only the subset of confirmed identities should be visible in the results returned by queries to the server.

Components: The client and the server are modeled as stateful classes which expose their functionality a set of operations. The server interface closely resembles that of the HAGRID API[5] and consists of operations for lookup,

[1] `https://verifythis.github.io`
[2] `https://gitlab.com/hagrid-keyserver/hagrid`
[3] `https://github.com/gernst/verifythis2020`
[4] `https://verifythis.github.io/2019-09-10-eventb/`
[5] `https://keys.openpgp.org/about/api`

requests for email validation and management access, and finally confirmation and deletion of associations between keys and identities. Internally, the server stores all keys every uploaded, the association between confirmed identities and keys, as well as the necessary bookkeeping for authenticated access in terms of tokens. A client object, on the other hand, stores a set of keys alongside those tokens received by the server, in order to be able to execute requests (e.g. in unit tests or randomly).

Communication: HAGRID uses two channels to communicate with users: a web-based interface for the lookup, upload, request for validation of keys, and management of these; and regular email for authentication tokens that are needed for certain operations. Messages are distributed by glue code in an actor-based approach, where the association between a client and its mailbox is explicit.

Adversary: We assume that all communication is secure. The adversary has ordinary access to all operations of the server but we assume that he/she cannot guess authentication tokens. As a consequence, the adversary has access to all information that is supposedly public, unless of course the server leaks secret or unconfirmed information.

3 Test Approach

We consider two kinds of properties of interest: functional correctness and security. In our setting, correctness means primarily that uploaded keys can be looked up successfully. Security, on the other hand, expresses that no secret or unconfirmed information is visible to the adversary.

We use property-based testing via ScalaCheck, a tool that enumerates inputs according to a generator schema, runs the system under test, and checks properties of the resulting states. For functional correctness, we might specify, for example, that a key can be successfully downloaded from the keyserver after an upload; and that certain previously confirmed identities in the key are present.

Dynamic monitoring of information flow is a little trickier. There are some existing approaches, such as tagging runtime values [1], and security type systems, such as Jif [9] for Java, and hybrid monitors that employ logical reasoning [4].

Our test approach is based on *Histories* that record events of the interaction explicitly, but not the internal states of the stateful actors. These histories are created by custom ScalaCheck generators with respect to a small collection of known email addresses. Three types of events can occur: upload of a key, and confirmation resp. revocation of the association between an email address and a key (via its fingerprint that is supposed to be unique).

The top-level specification is expressed over histories, which precisely determines those associations that are intended to be visible to another user of HAGRID (i.e. the adversary). To test whether an actual execution is correct and secure we thus compare the intended status of each *potential* association from the predetermined set of identities and keys to the actual observations that can be made by calling server operations.

Ongoing effort integrates this model with the actual implementation of HAGRID for cross-validation: We have written a back-end that dispatches the test calls directly over the REST API and reads back confirmation emails issued by HAGRID from a mailbox in the file system. Preliminary results have not show any odd behavior.

References

1. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Programming Languages and Analysis for Security (PLAS). pp. 113–124 (2009)
2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: Security and Privacy (S&P). pp. 339–353. IEEE (2008)
3. Callas, J., Donnerhacke, L., Finney, H., Thayer, R.: OpenPGP message format. Tech. rep., RFC 2440, November (1998)
4. Chudnov, A., Kuan, G., Naumann, D.A.: Information flow monitoring as abstract interpretation for relational logic. In: Computer Security Foundations Symposium (CSF). pp. 48–62. IEEE (2014)
5. Chudnov, A., Naumann, D.A.: Assuming you know: Epistemic semantics of relational annotations for expressive flow policies. In: Computer Security Foundations Symposium (CSF). pp. 189–203. IEEE (2018)
6. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM sigplan notices **46**(4), 53–64 (2011)
7. Goguen, J., Meseguer, J.: Security policies and security models. In: Security and Privacy (S&P). pp. 11–20. Computer Society, Oakland, California, USA (1982)
8. Lourenço, L., Caires, L.: Dependent information flow types. In: Principles of Programming Languages (POPL). pp. 317–328. ACM (2015)
9. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow (2001)
10. Zheng, L., Myers, A.C.: Dynamic security labels and static information flow control. International Journal of Information Security **6**(2–3) (2007)

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 preconditions 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:

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

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.

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

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:

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

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 $id \in k.ids$, 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.

VerifyThis Collaborative Long Term Challenge The PGP Key Server
– Challenge Manual –

Marieke Huisman[1], Raúl Monti[1], Mattias Ulbrich[2], and Alexander Weigl[2]

[1] University of Twente, Enschede, Netherlands
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany

1 Introduction

The VerifyThis verification competition is a regular event run as an onsite meeting with workshop character in which three challenges are proposed that participants have to verify in 90 minutes each using their favourite program verification tool.

We have experienced that the state of the art of program verification allows the participants to specify and verify impressively complex algorithms in this short a time span. If such sophisticated, realistic but not real, problems can be solved in real-time, what would be achievable if (a) we as the program verification community collaborated and (b) the time constraints were removed?

The *VerifyThis Collaborative Long Term Challenge* aims at proving that deductive program verification can produce relevant results for real systems with acceptable effort. This challenge is not competitive. It would be nice if one group could prove the protocol of the system to be correct whereas another group would show that an implementation follows that protocol, and is hence correct. Participants have time from mid August 2019 until end of February 2020 to choose aspects and parts of the challenge system that they try to specify and verify with the tool of their choice.

Section 2 introduces the target system that we want to verify. Section 3 lists a number of potential verification challenges and Section 5 contains the natural language requirements for the operations. Section 4 finally gives you hints on how to contribute to the challenge.

2 The OpenPGP Key Server

When using public key encryption and signatures in e-mails, one challenge is to obtain the public key of recipients. To this end, public key servers have been installed that can be queried for public keys. The most popular[3] public key server OpenPGP was recently shown to have severe security flaws. There was no protection on who could publish a key for an e-mail address and no protection

[3] It is the default server used by the Thunderbird public-key engine *Enigmail* for instance

on the amount of data published. This opened the gate for attacks: An attacker could publish a large number of large keys for the identity of the attacked. This led to two results: People, who want to send an e-mail to the target, might choose an untrusted key that does not belong to the target, but someone completely different.[4] They would also not be able to pick the right key entry from the key server amongst the many fake ones. And more critically, clients (like the GPG) of the service , have struggle to handle these large *spam* keys—resulting into CVE-2019-13050. More background information on *denial-of-service* attacks is available in the blog post of the developers. Moreover, the old key server software SKS did not conform to the General Data Protection Regulation (GDPR) and had performance issues.

As a consequence, the OpenPGP community decided to implement a new server framework that manages the access to public keys. The new official server is called HAGRID, it is open source [5], and it is already in production. HAGRID is written in the programming language Rust and comprises some 6,000 lines of code in total[6]. This implementation is the *reference implementation* of a *verifying* key server.

The server is essentially a database that allows users to store their public key for their e-mail address, to query for keys for e-mail addresses and to tracelessly remove e-mail-key pairs from the database. To avoid illegal database entry and removal actions, confirmations are sent out to the e-mail addresses of issuing users upon an addition or removal request.

The server possesses a web frontend which accepts requests from users or via restful API. It additionally possesses a connection to a database from which it reads key-value pairs and writes to it, and a channel for sending e-mails. At the core of the server there are four operations that can be triggered from outside the server via HTTP-API-requests to the web frontend. The operations are:

Request adding a key A user can issue a request for storing a key for a particular e-mail address. To avoid that anybody can store a key for someone else's e-mail address, the key is not directly stored into the database, but stored intermediately. The user retrieves confirmation code via the given e-mail to verify the specified address. Only once the confirmation code is activated, will the address be actually added to the database.

Querying an e-mail address Any user can issue a request for learning the key(s) stored with a concrete and verified e-mail address. Unlike on the old public server, queries for patterns are not allowed on the HAGRID server. Public keys that have been (verified) removed or have not yet been confirmed must not be returned in queries.

Request removing a key The user can request the removal of the association between a key and an e-mail address. The process begins with the confirmation via the e-mail address: The user enters one of their previously

[4] There is a security mechanism called the web of trust that should prevent one from using such untrusted keys.

[5] Available at f `https://gitlab.com/hagrid-keyserver/hagrid`

[6] Not including the underlying web framework or GPG library code

confirmed addresses. The server sends an e-mail to this address containing a link. Behind this link, there is a website that allows the removal of the key's association.

Confirming a request Additions and removals are indirect actions. Instead of modifying the database directly, they issue (secret and random) confirmation code. Confirmation of a code is performed using this operation. If the provided code is one recently issued then the corresponding operation (addition/removal) is finalised.

Section 5 contains natural language requirement specifications of these operations of the server.

3 Challenges

The challenge is to prove that a key server application is correct. This may be done by analysing the existing Rust reference implementation, by abstracting from the code, or by re-implementing the requirements of the key server in an own implementation.

Instead of verifying the reference implementation, any system that implements requirements from Sect. 5 can be considered for verification. The part of the key server to be considered the core system is also defined there. An implementation may make use of underlying (provenly or assumedly correct) libraries and middleware for the database or e-mail handling or webserver management.

On the other hand, if you are up to a larger challenge, go ahead and extend the scope of your verification and include (parts of) the webserver frontend and/or the database backend!

The *Collaborative Long Term Challenge* proposes a number of concrete verification missions (sub-challenges so to speak) that allow participants to shape their verification effort. They are meant as a guideline for addressing the challenge and as a starting point for dicussions on what should actually be verified about a system and using which technology. However, there are certainly interesting and critical missions not among the ones mentioned in this document which could (and should) be formally analysed. Any participant is explicitly encouraged to add to the long term challenge by contributing with additional missions – and possible answers for them.

Mission 0 (Identify relevant properties) *Identify properties of the key server that are worth being formally analysed. Formalise the properties in a formalism of your choice and verify them using the tool of your choice.*

Discuss the relevance of the properties (e.g., security, safety, performance, . . .).

The first suggested mission of the *Collaborative Long Term Challenge* is the least specific task and allows almost many formal code analysis tool to participate. Verification tools that run fully automatically (like the participating tools in the VSCOMP[7] competition series) are particularly invited to contribute solutions to this challenge and to show that results can be obtained without further user input. For this foundational verification question, no formal (full) specification is required.

Mission 1 (Safety) *Verify that the implementation of the key server does not exhibit undesired runtime effects (e.g., no runtime exceptions in Java, no undefined behaviour in C, . . .).*

Traditionally, the challenges in VerifyThis are more heavy weight, with concrete application-specific requirements that go beyond safety conditions and assertion checking. They strive to establish properties that require a logical formalisation against which the code needs to be verified. Depending on the complexity of the code and specification (and technique), the verification may then run automatically, or (in many cases) requires some form of user guidance (on top of the specification).

Unlike Mission 1, this *functional* verification requires knowledge about what the system has to compute. To this end, this document features a natural language description of the requirements of the core operations in Sect. 5. They are to be taken into consideration for the next mission and must be made accessible in the formalism of the particular formal verification approach you are using.

Mission 2 (Functionality) *Formalise the natural language specifications from Sect. 5 for the core operations.*

Prove that the implementation of the operations satisfy your formalisation.

One example functionality property is that if an e-mail address is queried, a key stored for this e-mail address is returned if there is one in the database.

Typically, functional verification is performed by formulating and proving *contracts* according to the design-by-contract paradigm. In other approaches, systems are specified using protocols or symbolic/abstract machines. This is particularly the case for model checking approaches where properties of the protocol can then be analysed on the abstract level.

Mission 3 (Protocol) *Encode the required key server behaviour as a formal state machine (automaton, protocol, state chart, . . .).*

Prove that the implementation adheres to this formal protocol.

Mission 3 is particularly well suited for a collaborative verification effort: One team of participants may prove properties of the protocol using an approach designed for that purpose (e.g., model checking), whereas another team verifies that the implementations of the operations adhere to their abstractions.

The last missions are classically called functional verification – and thus ressemble to the onsite challenges of the VerifyThis competition. Feel free to make adaptations to the specifications, make them as strong as possible. You are

[7] See `https://sv-comp.sosy-lab.org`

invited to add to your contribution a discussion of why this formal specification captures the informal properties correctly; or why it deviates from it.

There are properties that cannot be formalised as functional properties. One example of these are privacy properties. The *Collaborative Long Term Challenge* shall not be bound to stop at functional properties, but you are invited to go beyond if your tools support it:

Mission 4 (Privacy) *Specify and prove that the key server adheres to privacy principles. In particular: (a) Only exact query match results are ever returned to the user issuing a query. (b) Deleted information cannot be retrieved anymore from the server.*

One example of such a property is that if an e-mail address has been deleted from the system, no information about the e-mail address is kept in the server.

Another field of interesting properties (that also have been addressed in VerifyThis recently) are questions around concurrency. They are centered around the actual implementation. One interesting property among the ones dealing with concurrency is

Mission 5 (Thread safety) *Specify and verify that the implementation under consideration is free of data races.*

Let us close this section by listing a few more missions that might inspire you when thinking about the challenges that you will tackle with the system.

Mission 6 (Termination) *Prove that any operation of the server terminates.*

Mission 7 (Randomness) *Prove that any created confirmation code is*

(a) randomly chosen (i.e. that every string from the range is equally likely),
(b) cannot easily be predicted,
(c) is never leaked, but as the return value of the issuing operation.

4 Contributing

This section gives you a number of hints on how you can contribute to the collaborative long-term verification challenge.

The time line This long term challenge is be open from mid August 2019 until end of February 2020. Results will be presented at the VerifyThis workshop at ETAPS 2020 in Dublin, Ireland. A call for papers for a special issue of a relevant journal is planned for that time.

Of course, your contribution to the challenge after this initial period is equally welcome and will also be published on the online resources.

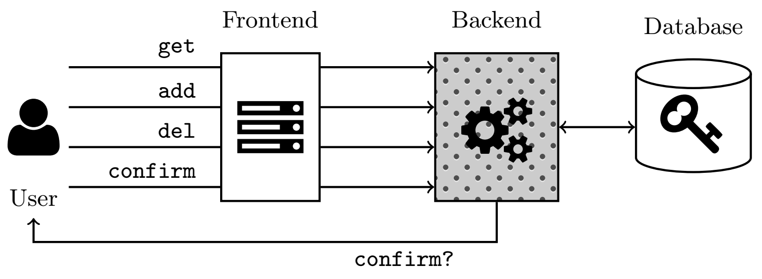

Fig. 1. Schematic of the key server architecture

The webpage The main entry point to the challenge and the main source of up-to-date information is the homepage of the long-term verification challenge:

`https://verifythis.github.io`

It is a collection of github-hosted pages that we will update frequently during the course of the challenge. All relevant resources will be linked on the page.

Moreover, this webpage will feature a collection of contribution sheets in which participants have indicated the properties that they consider for verification, the implementation they work on, the state of success, and more. This collection will be the point where participants can look for potential collaboration partners.

The mailing list We have set up a mailing list for the long-term challenge. Consider subscribing to the (moderated) mailing list at

`https://www.lists.kit.edu/sympa/info/verifythis-ltc`
`verifythis-ltc@lists.kit.edu`

if you want to be kept updated about the challenge and want to share questions and information with your colleagues.

5 Requirements

This section is likely to change over time as more experience with the implementation will have been gathered.
This is version 1 as of June 12, 2020.

The key server can be separated into three components: the *webserver* (frontend), the *key manager* (backend), and the key *database*; see Fig. 1 for a schematic of the architecture. The frontend is a HTTP-based webservice (REST interface) that receives incoming requests and sends the responses. Technical details on the actual OpenPGP API and protocol can be found on the server's webpage[8].

[8] `https://keys.openpgp.org/about/api`, accessed 2019-08-19

This challenge focuses on the key manager component of the server. This is a program that must provide implementations for the operations outlined in Sect. 2. For the sake of better accessibility of the program w.r.t. today's verification technology, the design of the operations of the key manager deviate a little from the actual implementation in the reference implementation, and delegate more work than typical to the web frontend. We mention some of the differences below, find more information on the server webpage (mentioned above). If the server in its simplified representation does not pose a (sufficiently difficult) challenge for you, you are invited to extend the example towards the real system, for instance by

(a) making it more performant by using sophisticated search-friendly data structures, or by
(b) implementing the actual API more closely, or by
(c) adding PGP key validation and address-extraction from keys, or by
(d) including (parts of) the web frontend or database backend into your endeavours.

The ultimate vision is to have a performant key server which is verified from web frontend all the way to the database.

The requirements are presented in form of tables that indicate the relevant facts. The basic types EMAIL, KEY, CONF-CODE are used to represent entities of the respective class of data (e-mail addresses, OpenPGP keys and confirmation codes, respectively).

5.1 General requirements

Like in reality, many requirements are implicit and go without mentioning here in this section. Examples are that the backend must not crash or that any request must terminate within in reasonable time.

When you choose to verify your own implementation, assumptions can be made about the way in which the key manager is invoked. If your language is object-oriented, for instance, it seems reasonable to assume that the key manager is a single object created at server startup that holds a reference to the key database and that handles all incoming requests. The operations would then be methods of the key manager class.

While this challenge primarly focuses on the key manager application, the webserver and the databse may as well be addressed in your verification.

More explicit general requirements may be added in later versions after discussions in the community.

5.2 Requirements for retrieving a key

Name	**get**
Parameters	e : EMAIL
Result	k : KEY $\cup \{\bot\}$
Precondition	none
Postcondition	If $k \neq \bot$, then the returned key k is associated with the given email address e in the database. $k = \bot$ iff there exists no entry for the given address e.
Effects	No changes on the database or pending (add or delete) confirmations.

Please note that this operation is deliberately kept indeterministic. If an e-mail address is associated to more than one key, then the operation may return any key k associated to e.

5.3 Requirements for adding a key

Name	**add**
Parameters	e : EMAIL, k : KEY
Result	c : CONF-CODE
Precondition	e and k are well-formed entities. e is an e-mail address to which the public key k applies. The tuple (e, k) may or may not already be present in the database or a confirmation for (e, k) may be pending.
Postcondition	The confirmation code c is unique[9] in the system. If (e, k) is present in the database, ... If a request is pending for (e, k), ...
Effects	The database remains unchanged. All pending confirmations are preserved. The only effect of the operation is that a confirmation request (c, k, e) may be added.

The actual protocol of the server is more complex. The addition function works as follows. Feel free to specify and verify the following code

```
void verifyingAdd(k: KEY) {
  emails = extractEmailAddressesFromKey(k);
  for( e : emails ) {
    token = add(e, k);
    if (token is valid)
```

[9] A confirmation code is unique iff it was previously not used in a pending add- or del-request.

```
      sendConfirmationEmail(e, token);
  }
}
```

The randomness of the confirmation code has not been mentioned here, but is an additional optional requirements.

5.4 Requirements for confirming a key

Name	**add!**
Parameters	c : CONF-CODE
Result	b : BOOL
Precondition	none
Postcondition	If the confirmation code c is valid and associated with a email-key pair (e, k), then (e, k) are confirmed and will be retrieved in future calls of **get(k)** until deletion. The confirmation code c becomes invalid after the first use. Return value b signals the success of this operation.
Effects	The existing entries in the database remain unchanged. All pending confirmations except associated one with c are preserved. Only (e, k) are added to the database if the confirmation code was valid.

5.5 Requirements for deleting a key

Name	**del**
Parameters	e : EMAIL, k : KEY
Result	d : CONF-CODE $\cup \{\perp\}$
Precondition	none
Postcondition	If e, k is a correct email and key and this pair is known in the database, then d is a valid unique confirmation code—otherwise, $d = \perp$.
Effects	The existing entries in the database remain unchanged. All pending add- and del-confirmations are preserved. Additionally, a new del-confirmation (c, e, k) is registered.

5.6 Requirements for confirming a key deletion

Name	**del!**
Parameters	c : CONF-CODE
Result	b : BOOL
Precondition	none
Postcondition	If the confirmation code c is valid and is associated with an email-key pair (e, k), then (e, k) are removed from the key database. The confirmation code c becomes invalid after the first use. Return value b signals the success of this operation.
Effects	The existing entries in the database remain unchanged, except the associated (e, k) is removed. All pending add- and del-confirmations, except the associated one with c, are preserved. The del-confirmation of c is revoked.