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

Marieke Huisman, Raúl Monti, Mattias Ulbrich, and Alexander Weigl

August 26, 2019

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[1] 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 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.[2] 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 [3], and it is already in

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

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

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

production. HAGRID is written in the programming language Rust and comprises some 6,000 lines of code in total[4]. 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 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.

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

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[5] 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 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.*

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

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.

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.

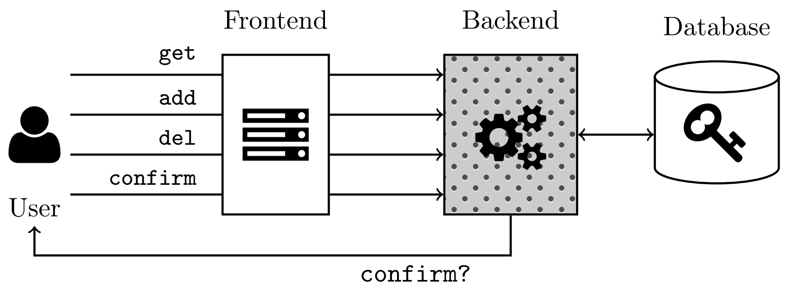

Figure 1: Schematic of the key server architecture

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 August 26, 2019.

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[6].

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 primarily 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.

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

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[7] 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)
      sendConfirmationEmail(e, token);
  }
}
```

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

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

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.