VerifyThis Collaborative Long Term Challenge
The PGP Key Server

Draft

August 19, 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 4 contains the natural language requirements for the operations. Section 5 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 not protection on who could publish a key for an e-mail address and no

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

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 *denial-of-service* attack are 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 a open source [3], and it is already in production. The project 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 the 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 API. It additionally possesses a connection to a database from which it reads key-value pairs and writes to it. And an e-mail connection.

At the core of the server there are n operations that can be triggered from outside via the web front end. These 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 a confirmation code is returned which is then sent by e-mail to 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 e-mail address. Unlike on the old public server, queries for patterns are not allowed on the HAGRID server. Public keys that have been 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 con-

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

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

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

firmation 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 then 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 a (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 4 contains natural language requirement specifications of these operations of server.

3 Challenges

The challenge is to prove such a real/realistic key server application correct. This may be done by analysing the Rust reference implementation or by abstracting from it towards a transition system or by re-implementing the requirements of the key server in your own implementation.

Instead of verifying the reference implementation, any implementation that satisfies the requirements from Sect. 4 can be considered for verification. An implementation may also make use of underlying (provenly or assumedly correct) libraries/middleware for the database or e-mail handling or web server management. On the other hand, if you are up to a larger challenge, go ahead and verify the web server front end and the database back end, too!

This challenge proposes a number of concrete verification tasks that allow participants to shape their verification effort. However, there are certainly other interesting properties not mentioned in this document which could be analysed. Any participant should feel free to add to the long term challenge by contributing additional questions and possibly answers for them.

Challenge 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.*

If you like, discuss the relevance of the properties (e.g., security, safety, performance, . . .).

In the following, this document describes a number of basic verification challenges for *Collaborative Long Term Challenge*. You are invited to tackle them or define your own goals.

The first challenge is the least specific task and allows almost many formal system to participate. Verification tools that run fully automatically (like the participants in the VSCOMP[5] competitions) are particularly invited to contribute solutions to this challenge and to show that this can be done without

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

further user input. For this foundational verification question, no formal (full) specification is required.

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

Traditionally, the challenges in VerifyThis are more heavy weight, and usually go beyond pure safety conditions and strive to establish properties that require a logical formalisation that needs to be verified. In many cases, some form of guidance (on top of the specification) for the verification is required.

To this end, this document features a natural language description of the requirements of the five core operations in Sect. 4. They are to be taken into consideration for the next challenge and must be made accessible for formal verification. We assume that every operation

Challenge 2 (Functionality) *Formalise the natural language specifications from Sect. 4 for the five 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, the key stored for this e-mail address is returned if there is one. This challenge is classical functional verification – and thus corresponds to the onsite challenges of the VerifyThis competition. Feel free to make to your specification as strong as sensible. You are invited to add to your contribution a discussion of why this formal specification captures the informal properties correctly.

There are properties that cannot be formalised as functional properties. One example of these are privacy properties. The long term challenge shall not be bound to stop at function properties, but can go beyond if tools support it:

Challenge 3 (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 amongst the ones dealing with concurrency is

Challenge 4 (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 challenges that might inspire you when thinking about the challenges that you will tackle with the system.

Challenge 5 (Termination) *Prove that any operation of the server terminates.*

Challenge 6 (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 Requirements

This section has not been completed yet.

5 Contributing

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

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

Of course, contribution to the challenge after this time 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, ... 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.