LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUTE FOR INFORMATICS

Software and Computational Systems Lab

Information Flow testing of a PGP server

Bachelorarbeit

Bearbeiter: Lukas Johannes Rieger
Supervisor: Prof. Dr. Gidon Ernst

Dokument erstellt
4. Oktober 2020

Statement of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgements. This applies also to all graphics, drawings, maps and images included in the thesis.

Place and date	Signature

Information Flow testing of a PGP server

Abstract

Lorem Ipsum

Contents

1 Introduction

Throughout the last decade, the advancements of digital technology have given rise to many novel ways of organizing and sharing information on a global level. Reaching from simple email applications to large social media platforms, these advancements have fundamentally changed how we interact with information and especially with each other. And yet, while many of these applications have undoubtedly had a largely positive impact on our productivity and everyday life, they have also sparked concerns about the privacy and authenticity of the information within these applications. **TODO: Verify2020 must be mentioned here!**

In this thesis, we formalize the claimed privacy features of one such solution – a public PGP server implementation called Hagrid – verifying the validity of these claims by running tests both on an abstract model as well as the concrete implementation. Chapter 1 will give a short introduction to the domain of public PGP servers by summarizing their internal structure and general behaviour. In Chapter 2, we then go on to address the origin and development of Hagrid. Next, Chapter 3 focusses on the theoretical background of information flow and briefly discusses a number of central terms that have been used in the past to classify and analyze different kinds of information flow. Chapter 4 then handles the central aspect of this thesis, which is the formal description and explanation of the privacy constraints defined by Hagrid. Chapter 5 then goes on to describe our concrete test implementation of the constraints laid out in Chapter 4. Chapter 5 details several related approaches specifically in the context of the VerifiyThis2020 challenge, that focus on information flow testing of Hagrid. Chapter 6 then ends with a short conclusion, reflecting on the work in this thesis.

[1] [2] [3] [4] [5] [6] [8] [9] [10] [13] [11]

2 Structure of a PGP server

2.1 How PGP servers are used

2.2 The Hagrid key server

Hagrid is a modern PGP key server that has been available since June 2019 under `https://keys.openpgp.org`. Initially, it was not intended to be a real replacement of existing key servers such as SKS - available at `https://sks-keyservers.net/`) - but was created to be used as a testing tool in various experiments for a new implementation of the OpenPGP standard under the name of "Sequoia".

Sequoia has been in development since early 2018 and is intended to be a performant and modern implementation of the OpenPGP standard written in the Rust programming language.

As outlined in one of the early developer blogs of Sequoia, concerns about the future and reliability of existing solutions like SKS and questions about how recent data protection regulations such as the GDPR would be addressed by these solutions finally lead to Hagrid evolving into a serious project aiming to provide a reliable key server implementation based on the Sequoia at its core.

How Hagrid avoids the mistakes of the past Besides the already mentioned concerns about the effects of recent changes in data protection laws, Hagrid also attempts to address several longstanding technical shortcomings of SKS that to this date remain largely unaddressed.

In an article announcing the official launch of Hagrid at `https://keys.openpgp.org`, the maintainers go on addressing some of these issues. For example, it is noted that the initial choice of implementing SKS in "unidiomatic OCaml" has proven problematic, as this has since then severely limited the number of maintainers willing to contribute to the project.

Further, they explain that the mental model that users may have of a PGP key server being comparable to a "telephone book" or indexable directory may oftentimes lead to confusion on the user side. This stems largely from the fact that implementations like SKS can effectively be seen as a append-only log of key updates without any process of verification.

Preventing certificate flooding through verification A specific example of the shortcomings of SKS is described in CITE SEQUOIA BLOG JULY 2019. As laid out in the article, in July of 2019 an unknown attacker was able to upload approximately 150000 signatures to the certificate of a well-known member of the OpenPGP community, therefore "poisoning" said certificate. As a result, any user that accidentally downloads one of these poisoned keys from an SKS key server will suddenly have to download a surprisingly large amount of completely useless data without any means of preventing it.

Arguably, this is the most important problem that Hagrid has set out to solve. The proposed solution lies in the fact that Hagrid is a *verifying* key server. Specifically, that means that Hagrid will only publish identifiable information that has been confirmed by the user through a specific process. As mentioned

in [CITE SEQUOIA BLOG] this also better matches the mental model of many users as described in chapter 2.2.

In this case, verification is implemented as a simple challenge-response scheme, in which the server will send a mail to all addresses belonging to the user IDs of some uploaded key. The user may then decide to follow the link contained within the mail, which will subsequently complete the verification process for the associated user ID. Until the server receives this final verification, it will strip the unconfirmed user ID from its associated PGP key.

While the developers of Hagrid recognize, that this mechanism doesn't provide any kind of *strong* verification, they nonetheless argue that their solution at least prevents enough unsophisticated vandalism and spamming (such as the previously mentioned SKS spam attack in July 2019), to justify this additional verification step.

Additionally, Hagrid is also capable of handling large PGP keys without causing a significant slowdown of the entire system. In contrast to that, alternatives such as GnuPG appear to come almost to a complete halt even for operations on unrelated keys as soon as any such flooded key has been imported into the keyring.

3 Information Flow Theory

The following sections give a brief introduction to the general notion of *security* in regards to the concept of information in computational systems. Building upon this, we then focus specifically on the key concepts of non-interference and declassification as they relate to the notion of information flow.

3.1 Security models and policies

Many modern computational systems will at some point have to handle certain kinds of information, which must never be accessible to an *unauthorized* entity. A popular description for a system, which keeps its information from leaking to the outside or any unintended entity/process can be found in the term of a *secure* system. In this context, "security" or "secure information flow" means, that "no unauthorized flow of information is possible" [7].

While such a general definition may be relatively easy to arrive at, actually *enforcing* the necessary constraints on the numerous information pipelines and flows within even a single program is often no easy task at all. In order to properly define which information may flow in what way within a part of a program or between programs, a common approach is to specify the individual components, between which a flow of information may occur and consequently assigning certain security *levels* to these individual constituents. This approach can for example be found in [7] by Denning, in which the concept of *security classes SC* is introduced. These classes can then be bound to any information receptacles or "objects" in order to denote their respective security clearance.

This chapter is way too short at the moment.

3.2 Non-interference and Declassification

An important concept in terms of secure systems that can be operated by multiple users either simultaneously or sequentially, is the concept of *non-interference*.

> One group of users, using a certain set of commands, is noninterfering with another group of users if what the first group does with those commands has no effect on what the second group of users can see

Mention, that the lattice model is very widespread and fundamental. Tie into this later aswell. (Public/Private can be seen as a binary lattice model.)

To explain this concept in terms of our problem domain of PGP servers, let us assume running instance of HAGRID which is publicly accessible. Let us further assume that there are two users $u1$ and $u2$ that are interacting with the server at the same time. Additionally, each user holds a certain set of keys:

$$u1 := \begin{cases} k1 = \text{Key(id1,id2)} \\ k2 = \text{Key(id3,id4)} \end{cases} \quad \text{and} \quad u2 := \begin{cases} k3 = \text{Key(id5,id6)} \\ k4 = \text{Key(id7,id8)} \end{cases}$$

Based on this, we can image the following, highly simplified interaction between the two users and HAGRID

The second call of $u2$ in this sequence is a key request for identity $id1$ which belongs to the previously uploaded key $k1$. Yet, because the identity had not

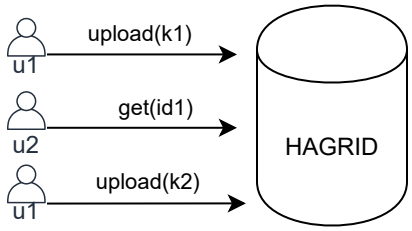


Figure 1: A non-interfering interaction

been verified when $u2$ issued its key request, the resulting response will be empty. That means, that the actions of $u1$ hat no effect *at all* on the experience of $u2$. As a matter of fact, based on the interactions with HAGRID alone, $u2$ has no way of identifying whether there even *is* another user currently interacting with the same system. In contrast to that, let us assume that this sequence of interactions is continued in the following way:

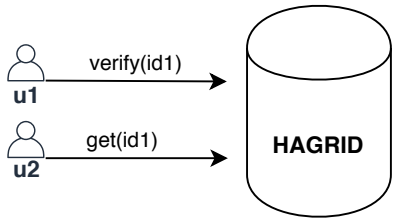


Figure 2: The interaction has become interfering

The second *get* request now yields a different response as before. Instead of an empty response, $u2$ now receives a key with an identity. This change in behaviour was caused by $u1$ issuing a verification for $id1$. Therefore, the actions of $u1$ were able to influence the sequence of interactions between $u2$ and HAGRID, which causes this interaction to no longer be *non-interfering*.

Definition of non-interference by Goguen and Meseguer A more formal definition of non-interference is given by Goguen and Meseguer in "Security Policies and Security Models", which is also the paper that initially popularized the notion of non-interference. In order to give a brief summary of their definition of non-interference, let us first introduce the central terms on which their definition relies. From here on, let Q denote some abstract system, which can be approximately modeled as some state machine $M = \{U, S, C, \text{Out}\}$. In this context, let U be a set whose elements denote the users of Q. Additionally, let S be the set of abstract states of M, while C denotes the set of "commands" that may cause a state transition in M. Finally, let Out refer to the set of all

"outputs" that Q may generate. Given these foundational definitions, Goguen and Meseguer proceed to define a function p:

Given a subset G of all users U and some subset A of state altering commands C, let w be in $(U \times C)^*$. That is, w is an ordered sequence of pairs consisting of a user and an associated command.

Finally, let $p_G(w)$ be equal to w excluding any pairs (u, c), where $u \in G$. Similarly, let $p_A(w)$ be equal to w excluding any pairs (u, c) where $c \in A$. In combination, let $p_{G,A}$ be equal to w excluding any pairs (u, c) where $u \in G$ and $c \in A$. From there, Goguen and Meseguer then directly derive three notions of non-interference:

1. Non-interference between two sets of users: Given a state machine M and two sets of users G and G', we let $G :| G'$ denote that G and G' are non-interfering, iff
$$[[\ w\ ]]_u = [[\ p_G(w)\ ]]_u$$
That is, they are only non-interfering if the outputs caused by w are equal to those produced by the subsequence of w excluding any element involving a user from G.

2. Non-interference between an ability A and a set of users G: Given a state machine M, we let $A :| G$ denote that A and G are non-interfering, iff
$$[[\ w\ ]]_u = [[\ p_A(w)\ ]]_u$$
That is, they are only non-interfering if the outputs caused by w are equal to those produced by the subsequence of w excluding any element involving an ability from A.

3. Non-interference between a set of users G with ability A and another set of users G': Given a state machine M, we let $A, G :| G'$ denote that no user from G with ability A interferes with users from G', iff
$$[[\ w\ ]]_u = [[\ p_{G,A}(w)\ ]]_u$$
That is, they are only non-interfering if the outputs caused by w are equal to those produced by the subsequence of w excluding all elements involving a user from G and ability from A.

Note that all of the definitions above describe purely *static* non-interference policies. That is, the non-interference policy must always hold and thus never allows any sets of users and abilities to interfere *at all*.

Given this definition of non-interference, it can be trivially concluded that strict enforcement of this principle would render most information systems practically unusable. To mitigate the strictness of these definitions, Goguen and Meseguer further introduce the notion of *conditional non-interference*, which encodes the notion that the definitions given above must only hold under some condition P:
$$G, A :| G' \text{ if } P$$

Applied to Hagrid: Let U be the set of all users. Let $G, G' \subset U$ be two sets of users that are both accessing the same key K. Let $u \in G$ and c be the set of state altering operations provided by Hagrid (Verify, Revoke):

$$\forall u, c \,.\, \{u\}, \{c\} :| \, G' \text{ if not } \text{CHECK}(u, c, K),$$

where $\text{CHECK}(u, c, K)$ describes whether u can use c to affect the state of K in any meaningful way.

That is, given some key K, we guarantee that a user u does not interfere with any other user that operates on the same key as long as u lacks the ability to verify or revoke identities associated with K. In return, that also means that we explicitly permit interference by supporting declassification of formerly confidential information in such cases, where it is permissible for a user to affect the privacy state of some key K.

As Sabelfeld and Sands explain in their paper "Dimensions and principles of declassification", the core issue therefore lies with ensuring that the release of information in these instances is actually *safe*. That is, that such a system never accidentally declassifies more information than it ought to, as that would consequently also violate its non-interference policies. The main challenge that one faces when trying to *verify* whether Hagrid adheres to these policies, is that the set of theoretically visible identities is constantly changing and strongly depends on dynamic runtime values within the Hagrid system. While there exist some approaches like the one laid out in "Dependent Information Flow Types" by Lourenço and Caires, which are based on Dependent Type Theory and allow one to formulate Information Flow types that depend on runtime values, these approaches are relatively novel and unexplored.

In the following chapter, we instead present an approach to ensure the safe declassification of sensitive information during runtime that utilizes the notion of state machines similarly to Goguen and Meseguer. We then extend the approach in such a way, as to allow us to specify exactly what sensitive information is declassifiable (or potentially reclassifiable) by Hagrid based on the input of the system.

4 History based security specifications

Our approach must specifically be able to encode and evaluate *dynamically* changing privacy constraints on a certain subset of relevant information. Additionally, these pieces of information are also dynamic and constantly changing while our system under test is actively operating. This poses several challenges in regards to how we can specify these constraints over the whole body of information beforehand. Instead, we have decided to utilise a *history* based encoding scheme, that takes the sequential nature of privacy changes over several time steps into account.

4.1 What constitutes a history

We rely on a central data structure, a sequential *history* of abstract events, that each encodes a high-level operation depending on the system under test. Formally, we can thus define any history h as

$$h_n = \{ev_1, ev_2, \ldots, ev_n\},$$

where n denotes the size of the given history and any ev_i represents the i-th operation or *event* in the history. Given that the history itself is in no way responsible for the way in which it is interpreted, we impose no requirements on the internal structure of the event type.

4.2 Generating histories for HAGRID

Given our general definition of a history, we can now look at the content of histories that we use to test our model of HAGRID. As described in chapter 2.2, we want to be able to describe the process of verifying a given subset of valid identities belonging to a PGP key and similarly revoking these identities in a similar fashion. Note that these two operations, *verifying* and *revoking* identities are the central operations, that drive the dynamic declassification and reclassification of private data in our server. We therefore define 3 abstract types of events, that can be included in a history:

1. *Upload(k)* – Given some key k, this event describes the action of adding a fresh PGP key to the internal state of our server.

2. *Verify(ids,fingerprint)* – Given a set of identities, this event describes the action of *declassifying* these formerly private identities, given that they belong to the key identified by the corresponding *fingerprint*

3. *Revoke(ids, fingerprint)* – Given a set of identities, this event describes the action of *reclassifying* these formerly public identities, given that they belong to the key identified by the corresponding *fingerprint*

4.3 How to evaluate histories

Given a history of some arbitrary size, we now require a method of inspecting the sequence of events and compute the *effect* that each event in the history has on the privacy state space of our server. Specifically, our goal is to run a symbolic execution of a history, which ultimately computes the totality of valid visibility associations between keys and their identities. Note, that this symbolic history execution doesn't have to match the actual execution process, because we can omit the server model as our middle man and compute the privacy state for all keys directly.

A formal description of history evaluation Evaluation of arbitrary histories can be formally approximated by a set of state machines S_i. Each state machine tracks the visibility state of its associated identity i.

To show how this formalization works, let us first look at a simpler case where we only consider histories of events that affect a single key with only one associated identity.

We can then define a state machine S which is a quintuple $(\Sigma, Q, s_0, \delta, F)$, where:

- Σ is the input alphabet given by the set $\{\text{Upload}, \text{Verify}, \text{Revoke}\}$, corresponding to our history event types;
- Q is the finite set of states $\{\Theta, U, C, R\}$;
- s_0 is the initial state of any identity, denoted as S_0;
- δ is the state transition function, defined by the following state transitions:

$$\begin{aligned} \Theta &\xrightarrow{\text{Upload}} U \\ U &\xrightarrow{\text{Verify}} C \\ C &\xrightarrow{\text{Revoke}} R \\ R &\xrightarrow{\text{Verify}} C \end{aligned} \tag{1}$$

- F is the empty set of final states

The states of this state machine represent the privacy state after each transition, where a transition to U means, that the given identity has been uploaded, yet remains undiscoverable until further state changes. The C state occurs, when the given identity gets verified and therefore publicly available, whereas the R state describes the exact opposite, the given identity is private must first be re-verified to become available again.

We can now generalize this state machine to support histories containing arbitrary amounts of keys and identities. Let's consider the set of state machines

$\{S_{i,k}\}$, where all state machines share their set of states and initial state with S.

Given some event $E_{i,k}$ and some state $s \in Q$, we define $\exists s' \subseteq \delta_{i',k'}(E_{i',k'}, s)$ if, and only if $\exists S_{i',k'}.\, i' = i \wedge k' = k$. Finally, it is easy to see, that any event $E(t, f)$, where t is a set of identities can also be denoted as the set $E' = \{E_{t_i,f} | t_i \in t\}$. That means, that any event that affects multiple identities at once can be equivalently formulated as a sequence of the same event that each affects a single event from the original set.

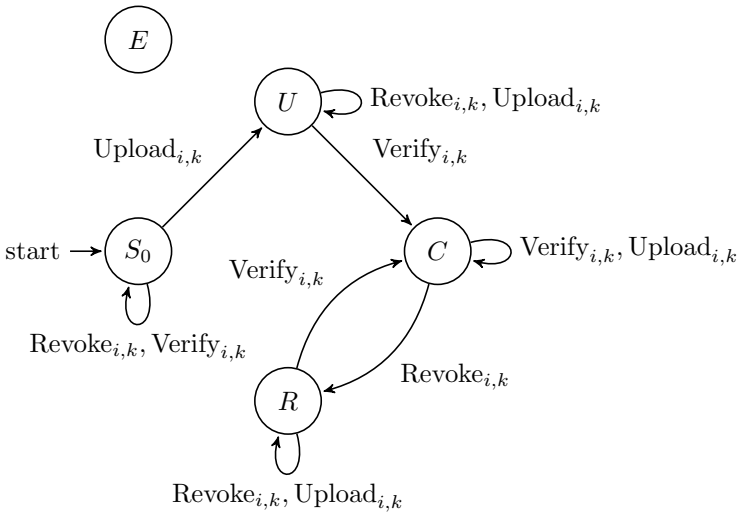


Figure 3: Visualization of a state machine over a history

In order to emphasize which states support a successful key request, we added an additional state E, which is only reachable by transitions marked with $\text{Get}_{i,k}$. We can think of E as a kind of implicit error state, which tells us, that the system encountered an attempt to retrieve a key k, while the state machine associated with k was still in a state, that made returning a key impossible.

It is easy to see, that this can only happen, if the state machine is either in its initial state S_0, or all state machines $t_{k,i}$ for a given k are currently in the state U.

Reformulating the example runs from chapter 3.2 using state machines Looking at our previous history examples from chapter 3.2 we can see quite clearly, that our encoding of histories in terms of state machines gives us a surprisingly natural representation of our privacy states at every intermediate step of our history. Let $h_1 = \{Upload(k1), Get(id1), Upload(k2)\}$ and $h_2 = \{Upload(k1), Get(id1), Upload(k2), Verify(id1), Get(id1)\}$.

We can now visualize both history runs accordingly, by showing the respective state transitions of their state machines at each step:

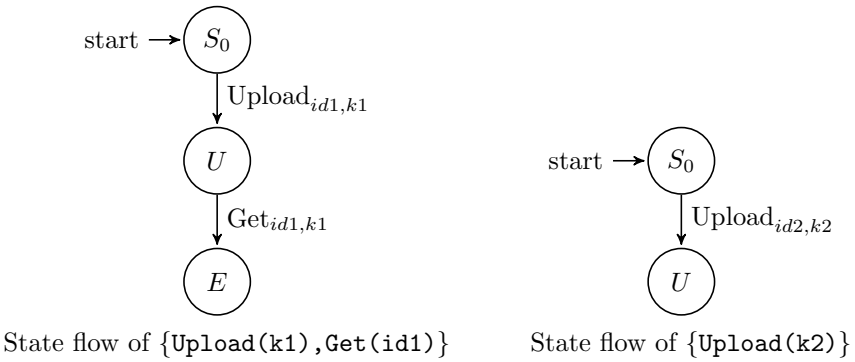


Figure 4: State flow of h_1 split up into two separate state machines

From this, we can easily determine that our set of publicly accessible information is empty for history h_1. Looking at the following state transition graphs for h_2 we can also see quite clearly how adding an instruction like `Verify(id1)` changes that:

As h_2 contains a sound {`Upload(...),Verify(...)`} sequence, the state machine associated with k_1 and id_1 ends in a state, that we have defined as belonging to a valid *public* state. We can thus easily determine *all* relevant privacy states by simply collecting the final state of each state machine.

Note that this encoding *only* works, because we can construct these state machines in a strictly separated manner. That means, that any event $E_{i,k}$ will only ever affect a single state machine and will not introduce any side effects involving several state machines.

TODO: Is this relevant?

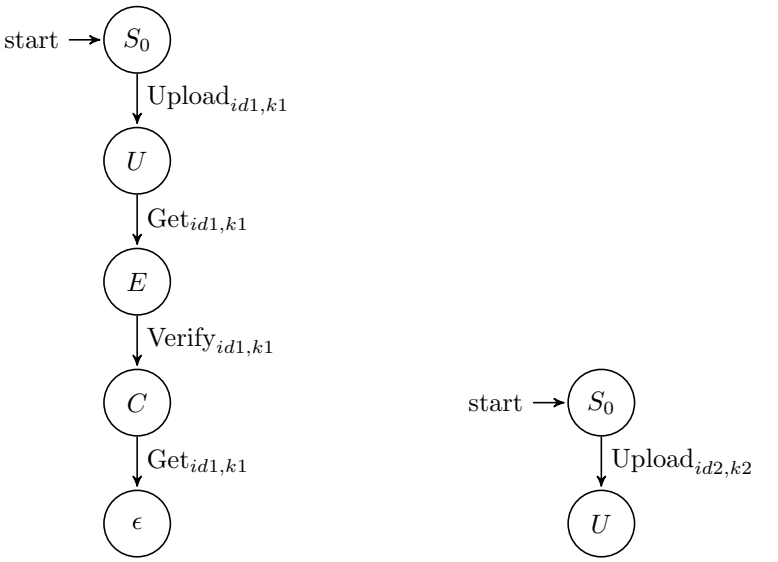
The first step of evaluating a history requires us to divide the data contained in the history into three distinct maps U, C and R, where U holds an entry for every uploaded key mapping the fingerprint to the key itself. C then maps each identity which remains *confirmed* at the end of the history to the fingerprint of its parent key. R on the other hand, maps all identities which remain revoked to its parent key.

Formally, the set of all uploaded keys can be constructed from a given history as follows:

$$U = \left\{ k \;\middle|\; \text{Upload}_{i,k} \in H \right\}$$

having $\text{Upload}_{i,k} \in H \Leftrightarrow \exists x.H_x = \text{Upload}_{i,k}$. From this set we can then define the total set of all revoked and verified `(Identity,Key)` pairs:

$$\begin{aligned} C &= \left\{ (i,k) \;\middle|\; \begin{array}{c} \exists t, t'.H_t = \text{Upload}_{i,k}, H_{t'} = \\ \text{Verify}_{i,k}, t < t', \forall z > t'.H_z \neq \text{Revoke}_{i,k} \end{array} \right\} \\ R &= \left\{ (i,k) \;\middle|\; \begin{array}{c} \exists t, t'.H_t = \text{Upload}_{i,k}, H_{t'} = \\ \text{Revoke}_{i,k}, t < t', \forall z > t'.H_z \neq \text{Verify}_{i,k} \end{array} \right\} \end{aligned} \tag{2}$$



State flow of {`Upload(k1)`,`Get(id1)`, `Verify(id1)`,`Get(id1)`}

State flow of {`Upload(k2)`}

Figure 5: State flow of h_2 split up into two separate state machines

Ultimately, we want to obtain a map, that associates the fingerprint of every uploaded key to a set of (`Identity`,`Status`) tuples. In this case, `Status` indicates the visibility of the associated identity, which may take either of the following three values:

- *Public* – The identity has been declassified and can be made publicly available.
- *Private* – The identity remains classified and must therefore not be leaked by the server.
- *Revoked* – The identity has been reclassified after having been confirmed at some prior point in the history.

TODO: formalize withUploaded, etc.

History evaluation against a server model Determining which associations between identities and keys are valid based on a history alone is only the first step of actually *testing* HAGRID. To complete our history-based approach, we require a way of comparing the results of the symbolic history execution to the actual server state. Only then can we determine, whether the model adheres to our chosen privacy constraints.

The solution to this requirement consists of two parts in total. As a first step, we need to *execute* the history on a given server backend. In a second step,

we can then collect all available data from the server and compare the results to our previously computed visibilities.

In order to visualize the execution of a history, we can look at the following example:

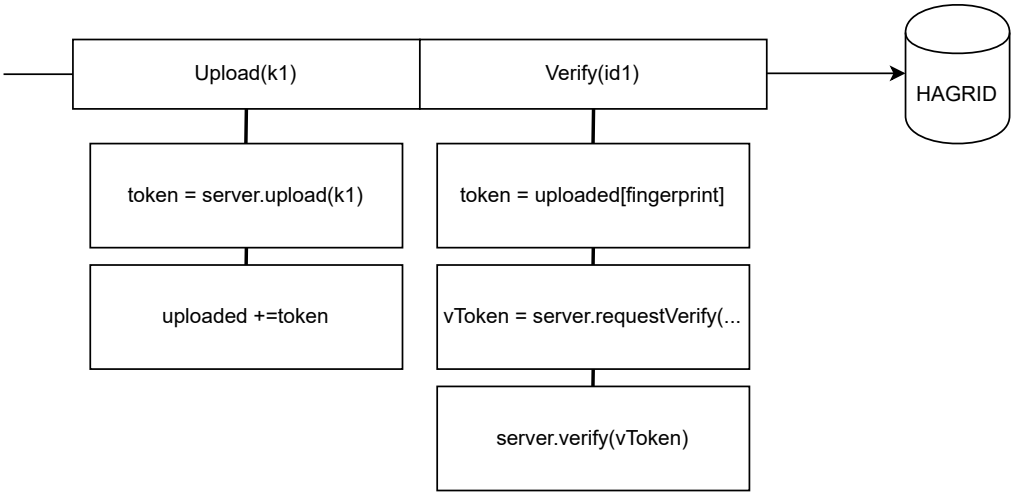


Figure 6: An example of a history execution

More examples here!

Notice, how `Verify` abstracts over the two separate steps of *requesting* a verification and the subsequent actual verification step. **TODO** The algorithm for testing HAGRID can then be summarized as:

1. Using all keys contained in the given history, send a key-request to HAGRID for every key k
2. Collect all returned identities in a map that associates each fingerprint with the corresponding set of identities: fingerprint $\rightarrow$ identities
3. For all keys contained in either the given history, or the server response: Compare the symbolic execution result $\{(\text{Identity}, \text{Status})\}$ with the matching server response $\{\text{Identity}\}$.

Determining invalid associations Given that we now have an abstract model to evaluate the valid associations of keys and identities based on a history of events, we can now determine whether some HAGRID backend adheres to its privacy model. This is done by computing the *difference* between the states that were determined to be visible by our history run and those returned by the backend.

For this purpose, we can abstract over the states of our state machines that directly expresses, whether the identity associated with this state machine is `Public`, `Private` or `Revoked`. The mapping from states to `Public` is very

straightforward, as it only consists of $\{C\}$. The same holds true for `Revoked`, which is represented only by the $\{R\}$ state. Finally, `Private` then encompasses the remaining to states $\{U, S_0\}$.

It should be noted that including a specific case for revoked identities would not be strictly necessary, as HAGRIDs returns an empty response both in the instance of `Private` and `Revoked` identities, rendering a distinction between these two states impossible from the outside. For the sake of accuracy, we still chose to include a `Revoked` category, because it can convey useful information in the hypothetical scenario where HAGRID *does* return a non-empty response and we want to include the exact kind of mismatch in an error message.

We can now use these three sets Public, Private and Revoked and determine the set of Valid and Invalid associations between keys and identities. This is done by retrieving all accessible information from the abstract server model and comparing each entry to the computed privacy state as shown in 2. Let S be the set of all `(Identity,Key)` pairs, representing the total retrieved state of our abstract server model. We can then define Valid as follows:

$$\text{Valid} = \left\{ (i,k) \;\middle|\; \begin{array}{c} ((i,k) \in C \land (i,k) \in S) \lor \\ ((i,k) \in R \land ((i,k) \notin S)) \lor \\ ((i,k) \notin (C \cup R) \land (i,k) \notin S) \end{array} \right\} \tag{3}$$

That is, for a given `(Identity, Key)` pair to be considered valid, it must satisfy one of three conditions:

1. The pair (i,k) is a confirmed association and is also included in the retrieved server state S
2. The pair (i,k) is a revoked association and is missing from the retrieved server state S
3. The pair (i,k) is neither revoked nor confirmed (but has been uploaded to the server) and is missing from the retrieved server state

Conversely, the definition of Invalid can be denoted as:

$$\text{Invalid} = \left\{ (i,k) \;\middle|\; \begin{array}{c} (i,k) \in C \land (i,k) \notin S \lor \\ (i,k) \in R \land (i,k) \in S \lor \\ (i,k) \notin (C \cup R) \land (i,k) \in S \end{array} \right\} \tag{4}$$

Again, for a given `(Identity,Key)` pair to be considered invalid, it must satisfy one of three conditions:

1. The pair (i,k) is a confirmed association but has no respective entry in the retrieved server state S
2. The pair (i,k) is a revoked association but is still present in the retrieved server state S
3. The pair retrieved is neither confirmed nor revoked (but has been uploaded to the server) but is still present in the retrieved server state

Based on these two partitions of identities and key, we can now define a general constraint that determines whether the Hagrid server model only performs valid declassifications (and reclassifications). Let H denote the set of all histories considered by a specific test instance. Let Invalid_h denote the set of invalid associations computed from some history $h \in H$. We then claim that a model of Hagrid is safe, if and only if the set of invalid associations is empty for all histories that were taken into account:

$$\forall h \in H \,.\, \text{Invalid}_h = \emptyset$$

Testing both functional correctness and privacy constraints at the same time During a presentation of our preliminary progress on the VerifyThis2020 challenge earlier this year in April, a member of one of the participating teams raised the question of how our approach handles a theoretically maximally secure HAGRID server, that never returns any key data of any kind. Such a system would achieve the highest possible protection of PGP keys because its behaviour would prevent it from ever leaking any information, private or otherwise. On the other hand, this behaviour would simultaneously render the server completely useless at its intended usage as a public PGP key server.

Given our formal approach of verifying privacy constraints as outlined in this chapter, we can strictly enforce the privacy of *unverified* identities, while simultaneously ensuring that the server correctly returns declassified data in all other cases, where the release of that information is actually permissible.

5 Practical implementation in Scala

The abstract server model In order to test the previously illustrated privacy constraints of PGP keys, we require a model of a PGP server which is sufficiently abstract to hide irrelevant implementation details, yet still exposes all functional specifications of the HAGRID server.

A minimal representation of a secure PGP server according the the HAGRID specifications would therefore require the following operations:

- *Request adding a key* - Request adding a new key to the database. This key may potentially contain an arbitrary amount of associated identities. The uploaded key is not made publicly available immediately after uploading. Instead, the server issues a confirmation code that may be used in subsequent confirmation requests.
- *Confirm an addition* - Confirm a previously uploaded `(Identity, Key)` pair given a valid confirmation code. This action does not directly confirm the selected identity but instead issues a confirmation email. The confirmation is only completed if the user uses the code contained within the email.
- *get(ByMail, ByFingerprint, ByKeyId)* - Retrieve a key from the database given some identifying index (e.g.the fingerprint of the PGP key or one of the associated identities).
- *Request a deletion* - Request the removal of some previously confirmed `(Identity, Key)` pair. This action does not directly delete the selected identity but instead issues a confirmation email.
- *Confirm a deletion* - Finalize the removal of some `(Identity, Key)` pair. This operation requires the confirmation code, which the user must have obtained from a confirmation mail issued by the previous operation.

This set of requirements allows for a relatively direct translation into an abstract base trait:

```
trait HagridInterface {
    def byEmail(identity: Identity): Option[Key]
    def byFingerprint(fingerprint: Fingerprint): Option[Key]
    def byKeyId(keyId: KeyId): Iterable[Key]
    def upload(key: Key): Token
    def requestVerify(from: Token, emails: Set[Identity]): Seq[Body]
    def verify(token: Token)
    def requestManage(identity: Identity): Option[EMail]
    def revoke(token: Token, emails: Set[Identity])
}
```

Source Code 1: Definition of the abstract server type

The datatypes `Fingerprint` and `KeyId` represent their PGP counterparts as defined in the OpenPGP message format according to [3]. The `Identity` type serves as a simple wrapper for a string containing an email address. `Key` combines the previously mentioned types into what our server model will treat as a representation of a full PGP key.

```
trait Key {
    def keyId: KeyId
    def fingerprint: Fingerprint
    def identities: Set[Identity]
}
```

Source Code 2: Definition of the abstract Key type

Given that we only needed a high-level representation of Hagrid, our implementation does not persist internal state. Instead, the model state is simply kept as a number of maps whose elements transition between each other:

```
class Server extends HagridInterface {
    var keys: Map[Fingerprint, Key]
    var uploaded: Map[Token, Fingerprint]
    var pending: Map[Token, (Fingerprint, Identity)]
    var confirmed: Map[Identity, Fingerprint]
    var managed: Map[Token, Fingerprint]
    ...
}
```

Source Code 3: Representation of the internal server state

Changing the internal server state Any of the actions exposed by the `HagridInterface` (see definition in section 1) may modify these internal state maps. For example, we can inspect the implementation of the `upload` method

```
def upload(key: Key): Token = {
    val fingerprint = key.fingerprint

    if (keys contains fingerprint)
    assert(keys(fingerprint) == key)

    val token = Token.unique
    keys += (fingerprint -> key)
    uploaded += (token -> fingerprint)
    token
}
```

Source Code 4: Implementation of `upload()`

we can see that both the `keys` map, as well as the `uploaded` map is being modified.

Invariants Internally, we defined several consistency invariants to ensure that our server model always behaves correctly. Any state manipulating action exposed by the server will trigger a re-evaluation of said invariants.

1. A key is valid if its fingerprint is registered in the `keys` map.

```
for ((fingerprint, key) <- keys) {
assert(key.fingerprint == fingerprint)
}
```

2. Upload tokens must refer to a valid fingerprint that itself refers to a valid key (see invariant 1)

```
for ((token, fingerprint) <- uploaded) {
    assert(keys contains fingerprint)
}
```

3. Pending validations must always refer to a valid key. Additionally, all pending validations must be for identity addresses that refer to the respective key.

```
for ((token, (fingerprint, identity)) <- pending) {
    assert(keys contains fingerprint)
    val key = keys(fingerprint)
    assert(key.identities contains identity)
  }
```

4. All confirmed identity addresses must refer to valid keys. Additionally, all confirmed identities must be valid for the associated key.

```
for ((identity, fingerprint) <- confirmed) {
  assert(keys contains fingerprint)
  val key = keys(fingerprint)
  assert(key.identities contains identity)
}
```

5. All issued management tokens must refer to valid keys.

```
for ((token, fingerprint) <- managed) {
    assert(keys contains fingerprint)
  }
}
```

5.1 Simulating actions and responses with histories

The server model itself cannot execute any concrete actions without some accompanying actor that causes this action. Generally, we wanted to express the following abstract actions in a composable way:

- *Upload some key* $\boldsymbol{k}$ *to the server*
- *Verify some identity* $\boldsymbol{i}$ *in relation to some parent key* $\boldsymbol{k}$
- *Revoke some verified identity* $\boldsymbol{i}$ *from its parent key* $\boldsymbol{k}$
- *Request some key* $\boldsymbol{k}$ *from the server, using one of its identifying characteristics*

Arbitrarily sized histories of events As outlined in chapter 4.3, we chose to denote the sequence of actions that should be simulated on the abstract server model through a "history" of abstract events. Through this approach, we can define a high-level model of *what* operations should be executed in a specific order while keeping the *how* - the concrete execution strategy - completely separate.

Once again, the different kinds of events are closely related to the capabilities exposed by our server model. Specifically, we recognise three distinct event types, that are modelled as subclasses of a `sealed trait Event`:

```
case class Revoke(ids: Set[Identity], fingerprint: Fingerprint)
    extends Event

case class Upload(key: Key) extends Event

case class Verify(ids: Set[Identity], fingerprint: Fingerprint)
    extends Event
```

Source Code 5: Event type implementation as defined in chapter 4.3

A complete history then simply acts as a lightweight wrapper around a buffer of `Event` objects:

```
case class History(events: mutable.Buffer[Event] = mutable.Buffer())
```

5.2 History evaluation strategies

Given some history h we want to determine precisely which associations between identities and keys should be visible depending on the combination of events in the history. To determine the visibility of an identity at any time, we introduce a `State` type, that tags the current state of an identity throughout the evaluation of the history. The state of an identity may take any of these three forms:

- Public: The identity has been confirmed and therefore publicly visible

- Private: The identity has not been confirmed yet and therefore remains private
- Revoked: The identity had been confirmed at some prior point but has since been revoked.

In terms of visibility, a `Private` identity and a `Revoked` identity can be considered equal from the perspective of a user. In both cases, the identity should not be included in any key requests.

In order to determine these states for every possible identity, we define a new method `def states: Map[Fingerprint, Set[(Identity, Status)]]` within `History`. The method starts off with three separate maps `Uploaded`, `Confirmed` and `Revoked`. These maps are then populated by folding over the sequence of events. Depending on the currently folded value, one or several of the maps will be updated.

```
val withUploadedAndConfirmed = confirmed.foldLeft(withUploaded) { (acc, elem) =>
    acc.get(elem._2) match {
        case Some(states) =>
            acc updated(elem._2,
                    states - (elem._1 -> Private) - (elem._1 -> Revoked) + (elem._1 -> P
                    )
        case _ => acc
    }
}.toMap
```

Source Code 6: Adding confirmed identities to their associated key

Executing a history on the server model Given a history and the computed privacy state based on that history, we can now define a simple execution strategy that sequentially loops through the history and sends the corresponding data to the server. The signature of the execution algorithm is given by `def execute(server: HagridInterface, history: History): Unit`. The algorithm then simply pattern matches the concrete event type and based on that type, calls the corresponding server method:

```
val uploaded: mutable.Map[Fingerprint, Token] = mutable.Map()
for (event <- history.events) {
  event match {
    case Event.Upload(key) =>
      uploaded += ((key.fingerprint, server upload key))
    case Event.Revoke(identities, _) =>
      for {
        head <- identities
        token <- server requestManage head map (_.token)
      } server revoke(token, Set(head))
```

```
      case Event.Verify(identities, fingerprint) =>
        for {
          uploadToken <- uploaded.get(fingerprint)
          Body(_, token, _) <- server requestVerify(uploadToken, identities)
        } server verify token
    }
  }
```

Source Code 7: Sequential execution of a history

As already mentioned in 4.3, the history itself is completely unaware of *how* it will be interpreted. In fact, the history is specifically asbtracting over any concrete execution details like having to receive and respond to confirmation mails.

This explanation is currently pretty bad and unfinished.

Maybe illustrate this with a diagram? Visualize the maps and how an event can change one or more of them.

5.3 Generating arbitrary histories with ScalaCheck

In order to ensure that our dynamic testing approach covers the largest possible space of possibly valid or invalid interactions with HAGRID we were unable to rely on some handwritten (and therefore non-exhaustive) set of predetermined test cases. Instead, our goal was to be able to generate diverse kinds of histories of arbitrary length and content, which would allow us to minimize the risk of missing some obscure edge-case in the behaviour of HAGRID.

5.3.1 Generating test data with ScalaCheck generators

We achieved this by utilizing a property based testing approach, in which all our test cases depend on a *generator* of arbitrary histories. The basic functionality for generators is provided by ScalaCheck, which exposes an abstract type `Gen[T]` for some type `T`. This type can subsequently be used as a potentially infinite source of test data, over which we can then express some universally quantified property that this data should possess. For example, the following code encodes the property that the act of taking the square root of any integer times itself should result in the original integer value:

```
val propSqrt = forAll { (n: Int) =>
    scala.math.sqrt(n*n) == n
}
```

We can now check that this property actually holds true by calling `propSqrt.check`. ScalaCheck will now automatically check, whether this property holds true for some sufficiently large amount of integer values, which are provided by a built in integer generator. The result of executing the property check will confirm to us, what we already know – namely that this property is obviously not true for any negative integer value:

```
scala> propSqrt.check
! Falsified after 2 passed tests.
> ARG_0: -1
> ARG_0_ORIGINAL: -488187735
```

What this output tells us, is that the first value found to falsify our property was $n = -488187735$. ScalaCheck was then able to *shrink* this failing instance down to the much simpler case of $n = -1$.

5.3.2 Implementing a generator for histories

In order to ensure that all events within a single history refer to a consistent set of common PGP keys and identities, we first generate a context object that carries all relevant information:

```
class Context(val keys: Map[Fingerprint, Key])
```

```
def contextGen(keySize: Int, idsPerKey: Int)
              (implicit keyGen: Int => Gen[Key]): Gen[Context] =
    for {
        keys <- Gen.listOfN(keySize,keyGen(keySize))
        keyMap = (keys map (k => (k.fingerprint, k))).toMap
    } yield new Context(keyMap)
```

In a second step, our generator produces a random sequence of events, in which all necessary key data is provided by the current context.

```
def uploadEventGen(implicit context: Context): Gen[Event] =
    for ((_, key) <- Gen.oneOf(context.keys)) yield Event.Upload(key)


def verifyEventGen(implicit context: Context): Gen[Event] =
    for {
        (fingerprint, key) <- Gen.oneOf(context.keys)
        identities <- Gen.someOf(key.identities)
    } yield Event.Verify(identities.toSet, fingerprint)


def revokeEventGen(implicit context: Context): Gen[Event] =
    for {
        (fingerprint, key) <- Gen.oneOf(context.keys)
        identities <- Gen.someOf(key.identities)
    } yield Event.Revoke(identities.toSet, fingerprint)
```

Finally, we combine these separate event generators into a single generator that supports all three kinds of event types

```
def eventGen(implicit context: Context): Gen[Event] =
    Gen.oneOf(uploadEventGen, verifyEventGen, revokeEventGen)
```

The complete history generator then simply combines both context and event generators and wraps the generated sequences in a `History` object.

```
def historyGen(length: Int)
              (implicit keyGen: Int => Gen[Key]): Gen[History] =
    for {
        context <- contextGen(2, 2)(keyGen)
        events <- Gen.listOfN(length, eventGen(context))
    } yield History(events.toBuffer)
```

TODO: Explain role of keyGen: Int =¿ Gen[Key]

5.4 Initial approach to simulating actions through "actors"

Our initial approach to simulating user actions was based on fine grained *actors* that each encapsulated a specific capability. Using this approach, any of the aforementioned operations would be represented by a single actor that runs on a simple state machine. The actor trait is defined as following:

```
trait Actor {
    def canAct: Boolean
    def act(): Unit

    val inbox = mutable.Queue[Data]()
    def canReceive = inbox.nonEmpty

    def isActive = canAct || canReceive

    def handle(from: Actor, msg: Message)
    def handle(from: Actor, msg: Body)
    def send(to: Actor, msg: Message): Unit =
        Network.send(this, to, msg)
    def send(to: Identity, msg: Body): Unit =
        Network.send(this, to, msg)

    def register(identity: Identity): Unit =
        Network.register(identity, this)
}
```

An actor could define a custom `act()` method that would initialize the specific action of this actor. Any future interaction with another actor would then be handled by the actors `handle()` method.

TODO

: Describe why there were two different handle methods. What is Network? What was bad about this approach? Show execution strategy and why order of execution was a problem.

6 Testing the real-world implementation of HAGRID

Given that we had successfully developed a working testing framework for our abstract model, we decided to integrate the actual HAGRID server into our testing efforts. For obvious reasons, testing HAGRID requires a local server instance to be running on the same machine.

Communicating with HAGRID The main effort of integrating HAGRID into our implementation focussed on the relaying and receiving of messages between our Scala frontend and HAGRID. To this end, we use *Sttp*, which is a simple library that enables us to communicate with the server through RESTfull http calls. The transferred data is being encoded as *JSON*, for which we rely on *Circe* as a library.

HAGRID itself defines a specific interface, through which external applications may communicate with the PGP server. This interface, called *Verifying Keyserver (VKS) Interface*, exposes the following endpoints:

- `GET /vks/v1/by-fingerprint/FINGERPRINT`
- `GET /vks/v1/by-keyid/KEY-ID`
- `GET /vks/v1/by-email/URI-ENCODED EMAIL-ADDRESS`
- `POST /vks/v1/upload`
- `POST /vks/v1/request-verify`

The interface which actually communicates with the server exposes exactly the same methods as our model, due to it implementing the same `HagridInterface`.

Instead of keeping track of keys and identities in the frontend, our `HagridServer` simply relays all actions to HAGRID by http. As an example, lets look at the `upload` method once again:

```
override def upload(key: Key): Token = {
    val response = basicRequest
        .post(hag("/vks/v1/upload"))
        .body(UploadBody(key.armored))
        .response(asJson[UploadResponse])
        .send()
        .body

    Token(response
    .toOption
    .get
    .token)
}
```

First, we construct a simple *POST*-Request which targets the *VKS* endpoint responsible for uploading PGP keys. We then attach the *armored* key-text as the request body and finally instruct the request to encode any eventual response as an instance of `UploadResponse`. The most crucial bit of information included in the `UploadResponse` is the token, which we require in order to send any verification requests at a later point.

However, it must be noted that our implementation, while fully working, generally *ignores* the possibility of an error occurring while communicating with HAGRID. We are aware that this may cause unforeseen problems in the future.

6.1 Receiving verification mails from HAGRID

While we were able to simulate the transmission of verification mails in an abstract and direct way for our server model, we had no such option when dealing with the real-world HAGRID server.

During the earlier development phases, we assumed that we would be forced to provide a dummy implementation of `sendmail`, as the documentation claimed that HAGRID would require a working `sendmail` command to be available in the environment in order to send and receive mails. Luckily, we discovered that HAGRID also provides an alternative mode of operation, in which all email transfer is handled directly through the filesystem. Modifying the configuration file `Rocket.toml` and adding the line `mail_folder=/PATH/TO/MAIL`, causes HAGRID to write all mails to a unique file in the specified folder.

This allows us to simply observe the given directory and react to any file modifications within the folder. More specifically, we defined a method, that would block its execution until it successfully read all expected mails:

```
  private def consumeMail(expectedSize: Int): Seq[Body] = {
      val watchKey = mailWatcher.take
      val events = watchKey.pollEvents.asScala
      val updatePaths = events
          .take(expectedSize)
          .map(_.asInstanceOf[WatchEvent[Path]])
          .map(_.context())

      val mails = updatePaths.flatMap { currentPath =>
          val resolved = mailPath.resolve(currentPath).toFile
          val source = Source.fromFile(resolved)
          val bodies: Seq[Body] = parseMail(source.mkString)
          source.close
          resolved.delete
          val isValid = watchKey.reset
          bodies
      }
      mails
}
```

where the returned sequence of mail bodies takes the form of:

```
case class Body(fingerprint: Fingerprint, token: Token, identity: Identity)
```

Parsing plain text confirmation mails In order to consume the emails within the selected directory and actually use the information contained within them, we had to parse the plain text content of said mails and extract the relevant information. In both cases of identity confirmation or revocation, these relevant bits of data would be the `Fingerprint`, the `Token` and finally the respective `Identity` in concern.

For example, a typical mail issued by HAGRID (running as a local instance) in repsonse to a confirmation request might look like this:

```
To: <ISSUE_EMAIL@ADDRESS>
Hi,

Dies ist eine automatische Nachricht von localhost.
Falls dies unerwartet ist, bitte die Nachricht ignorieren.

OpenPGP Schlüssel: 23B2E0C54487F50AC59134C3A1EC9765D7B25C5A

Damit der Schlüssel über die Email-Adresse "<ISSUE_EMAIL@ADDRESS>" gefunden werden kann,
klicke den folgenden Link:

http://localhost:8080/verify/vrTohvV8q552KMvARBE7foqkvGtUrfDl3iiyX9yqeOX

Weitere Informationen findest du unter http://localhost:8080/abo
```

This mail should be in english as well.

We currently use a regular expression to parse the relevant information from these mails. This functionality is summarized in a method `parseMail`:

```
val VERIFY_PATTERN: Regex =
    "(?s).*To: <(\\S+)>.*OpenPGP key: ||
    (\\S+).*http://localhost:8080/verify/(\\S+).*<!doctype html>.*".r

def parseMail(mail: String): Seq[Body] =
    decode[HagridMail](mail)
        .map(mail => new String(mail.message))
        .map {
        case REVOKE_PATTERN(identity, fingerprint, token) =>
            Body(FingerprintImpl(fingerprint), Token(token), PgpIdentity(identity))
        case VERIFY_PATTERN(identity, fingerprint, token) =>
            Body(FingerprintImpl(fingerprint), Token(token), PgpIdentity(identity))
        }
        .toSeq
```

As seen in the code sample in 6.1, this function is responsible for parsing the mails, that `consumeMail` read from the dedicated mail directory.

6.2 Generating valid PGP keys

While our testing efforts were mainly focussed on our own abstract model, we were able to use a simplified representation of our `Key` datatype to represent a PGP key.

```
sealed trait Key {
    def armored: String
    def keyId: KeyId
    def fingerprint: Fingerprint
    def identities: Set[Identity]
    def restrictedTo(ids: Set[Identity]): Key
}
```

For example, `keyId` simply consists of a unique string identifier that does not hold any specific meaning:

```
sealed trait KeyId {
    def value: String
}
case class KeyIdImpl(id: String) extends KeyId {
    def value: String = id
}
object KeyId {
    def random: KeyId = KeyIdImpl(UUID.randomUUID().toString)
}
```

In order to transmit any `Key` values to the actual HAGRID server, we couldn't rely on the existing dummy implementation, as HAGRID would reject any PGP key that is not properly formatted.

In order to solve this problem, we decided to implement a mechanism that is capable of generating valid PGP keys. For this purpose, we use *BouncyCastle*, which is a Java library for cryptographic use cases.

The general signature to generate a PGP key is as following:

```
def genPublicKey(identities: Set[Identity]): (PGPPublicKey, String)
```

It should be noted though that we currently do not support the generation of keys with arbitrary amounts of identities. This is due to several unsolved problems which we encountered during development. In the end, these hurdles led to the decision that temporarily supporting a maximum amount of only *two* identities is acceptable.

6.3 Initializing

One major challenge that we faced when trying to incorporate the actual HAGRID implementation in our testing approach was the fact that HAGRID maintains internal state across several distinct instances. This is due to HAGRID persisting uploaded keys and their contents within its installation folder. In contrast to this, our high level model of HAGRID maintains no state at all between any two separate test runs.

This is problematic, because our testing approach fundamentally relies on a limited amount of *Identities*, that are subsequently used to construct PGP Keys. This would necessarily lead to unexpected results when applying our approach to HAGRID, where there could potentially already be an identity associated to some PGP key, that was uploaded during a previous testing session. To elaborate on this problem, let us assume these two separate test runs in pseudocode:

Assuming that there exists a value `key1 = Key(identity1,identity2)` and a value `key2 = Key(identity3,identity4)` where all arguments to the `Key` constructor are instances of `Identity` with an arbitrary email address, we can define the following two test runs:

```
upload(key1)
verify(identity1)
```

Test run 1

```
verify(identity1)
upload(key2)
```

Test run 2

Looking *only* at test run 2 we would expect an outcome, in which the server returns no public keys/identities at all. Instead, when we actually execute these two runs in sequential order, we receive a PGP key when querying the server with `identity1`, even though we *never* uploaded it during the second test. Any solution to this problem therefore requires us to reset the internal state of HAGRID after having completed a test. This in turn also imposes an additional non trivial performance penalty on the execution speed of our ScalaCheck test, besides the slowdown caused by having to generate real PGP keys. For our current solution to this problem, we decided to compile HAGRID into a static image. This allows us to create a fresh instance of HAGRID at the beginning of each test iteration and shutting it down afterwards, at which point we simply clear the local directory of PGP keys. This ensures that each test run starts on a truly equal playing field.

7 Related Work

8 Conclusion

References

[1] Thomas H Austin and Cormac Flanagan. "Efficient purely-dynamic information flow analysis". In: *Programming Languages and Analysis for Security (PLAS)*. 2009, pp. 113–124.

[2] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. "Expressive Declassification Policies and Modular Static Enforcement". In: *Security and Privacy (S&P)*. IEEE, 2008, pp. 339–353.

[3] Jon Callas et al. *OpenPGP message format*. Tech. rep. RFC 2440, November, 1998.

[4] Andrey Chudnov, George Kuan, and David A Naumann. "Information flow monitoring as abstract interpretation for relational logic". In: *Computer Security Foundations Symposium (CSF)*. IEEE. 2014, pp. 48–62.

[5] Andrey Chudnov and David A Naumann. "Assuming You Know: Epistemic Semantics of Relational Annotations for Expressive Flow Policies". In: *Computer Security Foundations Symposium (CSF)*. IEEE. 2018, pp. 189–203.

[6] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *ACM sigplan notices* 46.4 (2011), pp. 53–64.

[7] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (May 1976), pp. 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056.

[8] Joseph Goguen and José Meseguer. "Security Policies and Security Models". In: *Security and Privacy (S&P)*. Oakland, California, USA: Computer Society, 1982, pp. 11–20.

[9] Luísa Lourenço and Luís Caires. "Dependent Information Flow Types". In: *Principles of Programming Languages (POPL)*. ACM, 2015, pp. 317–328.

[10] Andrew C Myers et al. *Jif: Java information flow*. 2001.

[11] Red Hat, Inc. *CVE-2019-13050: Certificate spamming attack against SKS key servers and GnuPG*. https://access.redhat.com/articles/4264021. Accessed: 2020-06-07. 2019.

[12] A. Sabelfeld and David Sands. "Dimensions and principles of declassification". In: July 2005, pp. 255–269. ISBN: 0-7695-2340-4. DOI: 10.1109/CSFW.2005.15.

[13] Lantian Zheng and Andrew C. Myers. "Dynamic security labels and Static Information Flow Control". In: *International Journal of Information Security* 6.2–3 (2007).