Formal Verification of Password Generation Algorithms used in Password Managers

Miguel Grilo
miguel.grilo@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal

Abstract

Password managers are important tools that enable us to use stronger passwords, freeing us from the cognitive burden of remembering them. Despite this, there are still many users who do not fully trust password managers. In this work, we focus on a feature that most password managers offer that might impact the user's trust, which is the process of generating random passwords. We survey which algorithms and protocols are most commonly used and we propose a solution for a formally verified reference implementation of a password generation algorithm. Finally, we realize this reference implementation in Jasmin and we prove that the concrete implementation preserves the verified properties. We use EasyCrypt as our proof framework and Jasmin as our programming language.
Keywords: Password Manager, Random Password Generator, Formal Verification, EasyCrypt, Jasmin.

1. Introduction

Passwords are still one of the most commonly-used means of user authentication. Although there are lots of research around password-based authentication [14, 7, 17, 19], there are still many problems and vulnerabilities regarding password authentication [9, 20, 15]. So, in general, security experts recommend using password managers (PMs) for both storing and generating strong random passwords [13]. Despite these recommendations, users tend to reject PMs partly because they do not fully trust these applications [1, 16, 4].

While trying to understand why users adopt PMs, different researchers observed that users consider the generation of random passwords as one important feature that makes them use these applications [1]. Moreover, users who do not use PMs with built-in password generators more often have weaker passwords and tend to reuse them [16]. Security experts have already warned about how dangerous it is to have weak passwords and, even worse, to reuse those weak passwords [12]. These studies suggest that a strong password generator that users can fully trust is a must-have feature for PMs, and guaranteeing that the generator is functionally correct and secure is an important concern.

1.1. Work Objectives

In this thesis, we have as main objectives the development of a formally verified reference implementation of a random password generator (RPG) and the implementation of it using Jasmin [2], both having to verified properties: functional correctness (generated passwords always satisfy the input password composition policy) and security (the probability of one password being generated is the same as any other password).

This reference implementation and the properties will be specified in EasyCrypt [5], an interactive framework for verifying the security of cryptographic constructions and protocols.

To understand the state of the art on random password generation we survey currently used and popular PMs password generators.

Regarding the Jasmin implementation, we want to integrate it into a PM, as a proof-of-concept of our solution.

2. Background

2.1. RPG Survey

As our first objective, we surveyed the algorithms and protocols for generating random passwords used by different PMs.

We studied a total of 15 PMs, but in this work we focus on three of them: Google Chrome's Password Manager (v89.0.4364.1), Bitwarden (v1.47.1), and KeePass (v2.46).

2.1.1 Password Composition Policy

All studied PMs allow users to define a password composition policy regarding the password to be generated. These policies define the structure of the password, and they are the input given to the RPG. Policy structures are really identical between the differents studied PMs. The user is, in all of them, available to define the length of the generated password and the set of characters that can appear in the password. Chrome and Bitwarden allow the user to define the minimum and maximum

occurrences of characters per set in the password, while in KeePass a user is able to define a character set "by hand" (i.e., choose any arbitrary set to sample characters from).

2.1.2 Random Password Generation

Almost all PMs use different algorithms to generate random passwords. However, the way they do it is very similar.

The main idea of these algorithms is to generate random characters from the union of the different character sets defined in the composition policy until the password length is fulfilled. To show how this is achieved, we present in pseudo-code (Algorithm 1) a generalization of the algorithms of the three PMs, being more closely related to Chrome's algorithm.

Algorithm 1 General Password Generation Algorithm

```
1: procedure GENERALGENERATE(policy)
2:     password ← ε
3:     foreach set ∈ policy.charSets do
4:         for i = 1, 2, ..., set.minOccurences do
5:             char ← Sample uniformly at random from set
6:             password ← password ||char
7:         end for
8:     end for
9:     while len(password) < policy.length do
10:        char ← Sample uniformly at random a char from the
       union of sets which have not yet reached their maximum
       number of occurrences of characters
11:        password ← password ||char
12:    end while
13:    Generate a random permutation of password
14:    Output password
15: end procedure
```

In general, these algorithms first iterate over all the sets from which they should sample from, according to the *policy*, and samples *minOccurrences* characters from each of them, appending them to the initially empty string *password*. Then, while the length of *password* is smaller than the *length* defined in the *policy*, the algorithm samples a character from the union of sets which have not yet reached their maximum number of occurrences of characters, and appends it to *password*. Finally, it outputs a permutation of *password*.

As said before, the previously described algorithm is essentially the same as Chrome's. KeePass's algorithm is simpler, since the user is not able to establish neither the minimum nor the maximum number of occurrences of characters per set. So there is no need to generate any string permutation, one just needs to define the single set of characters, which is the union of all available sets, and then randomly generate characters from that set.

Regarding Bitwarden, the only difference compared to the general algorithm described before is that is makes the permutation before generating the characters (i.e., it creates a string like 'lllluunl' to say that the first three characters are lowercase letters, the following two are uppercase letters, then a number, and finally another lowercase letter. Only then it generates the characters from the sets and places them in their respective position).

2.1.3 Random Character Generator

The process of uniformly sampling characters from a set is trivial if we possess a random number generator (defined next in Section 2.1.5). One just needs to index each character from the set (this comes automatically if we instantiate our sets as lists) and then we generate a number from 0 to the size of the set uniformly at random, and we output the character with that number as index.

2.1.4 String Permutation

Given the need to generate a random permutation of the characters of a string, Bitwarden and Chrome both implement the Fisher-Yates shuffle [8].

2.1.5 Random Number Generator

As we can see from Sections 2.1.3 and 2.1.4, the RPG needs to implement a Random Number Generator (RNG) have some source of randomness to generate the characters and a permutation of strings of characters.

After inspecting the code of the three PMs, we noticed that Chrome and KeePass had an identical RNG, because they generate numbers from 0 to an input *range*. The main idea of these two PMs is to generate random bytes, then casting them to an integer, and then return that value modulo range, so the value it generates is between 0 and range.

To generate random bytes, the three PMs considered had different approaches. Chrome uses different system calls based on the operating system the PM is running, Bitwarden uses NodeJS *randomBytes()* method, while KeePass defines their own random bytes generator based on ChaCha20.

On uniformity and maximum integer values. One must be careful with the approach described above because, since there is a finite maximum value for the random integer (which is $2^n - 1$ for n-bit words). This may lead to a non-uniform distribution over the possible values. For example, if we represent integers using 3-bit words (having the maximum possible value $2^3 - 1 = 7$) and if the input *range* is 5, then a result of 1 would be twice as likely as a result of 3 or 4.

So, some of the possible numbers generated by the random bytes generator must be discarded be rejection sampling. While this not guarantees termination, it allows the RNG to sample numbers uni-

formly, with a very high probability of terminating in very few tries. We explain Chrome's solution to this problem (KeePass is slightly different, but follows a similar idea).

In Chrome, the RNG first s the maximum 64-bit unsigned integer *maxValue* such that, given an input *range*, $maxValue \equiv range - 1 \bmod range$ (i.e., the greatest multiple of $range$ that can be written in 64 bits, subtracted by 1). Then, generates random values until it is generated a number which is smaller or equal to *maxValue*. Since this *maxValue* is the maximum value in these conditions, there is a high probability of generating a valid number in very few tries.

Chrome algorithm is described in pseudo-code in Algorithm 2.

Bitwarden RNG is more complex since it generates numbers from an arbitrary minimum value up to an arbitrary maximum value, so we will not go into detail.

Algorithm 2 RNG with maximum range

```
1: procedure CHROMERNG(range)
2:     maxValue ← (uint64.maxValue/range) * range − 1
3:     do
4:         value ← (uint64) GenerateRandomBytes
5:     while value > maxValue
6:     return value mod range
7: end procedure
```

While both Chrome's and KeePass's algorithms sample uniformly and are very efficient at doing it (i.e., reject a very small number of samples) they are still not ideal. We noticed that, for both RNGs, in some situations there are numbers being rejected that, if accepted, would still produce an uniform distribution (e.g., with Chrome's RNG, if we use 3-bit words to represent our integers (maximum value of 7), and if the input *range* is 4, this RNG will reject samples greater than 3 (since *maxValue* will take the value of (7/4)*4-1=3). So, from the interval $[0; 7]$ the algorithm only accepts integers in the interval $[0; 3]$, but could also accept integers from $[4; 7]$ and the distribution would still be uniform and the algorithm more efficient.

2.2. Technology Used

2.2.1 EasyCrypt

We use EasyCrypt to implement our RPG reference implementation and to reason about it. EasyCrypt [5] is an interactive framework for verifying the security of cryptographic constructions and protocols using the game-based approach. Cryptographic games and algorithms are modeled as *modules*, which consist of typed global variables and procedures written in a simple imperative language featuring random sampling operations.

Built around these modules, EasyCrypt implements program logics for proving properties of imperative programs.

Hoare Logic. It is a formal system composed of a set of rules for each syntactic construct of an imperative language used to reason compositionally about the correctness of programs. In EasyCrypt, an HL judgement has the form

```
hoare [M.p : A ==> B]
```

where `p` is a procedure of module `M`; `A` is an assertion (precondition) on the initial memory that may involve global variables of declared modules as well as parameters of `M.p`; `B` is an assertion (postcondition) on the final memory which may involve the term `res` which is the returned value of `M.p`, as well as global variables of declared modules.

The informal meaning of the HL judgment is that, for an initial memory &m satisfying `A`, if running the body of `M.p` in &m results in termination with another memory &m', &m' satisfies `B`.

Probabilistic Hoare Logic. Similar to HL, but allows carrying out proofs about the probability of a procedure's execution resulting in a memory satisfying a given postcondition. In EasyCrypt, a pHL judgement has one of the forms

```
phoare [M.p : A ==> B] < e
phoare [M.p : A ==> B] = e
phoare [M.p : A ==> B] > e
```

where `p` is a procedure of module `M`; `A` is an assertion (precondition) on the initial memory that may involve global variables of declared modules as well as parameters of `M.p`; `B` is an assertion (postcondition) on the final memory which may involve the term `res` which is the returned value of `M.p`, as well as global variables of declared modules; `e` is an expression of type real.

The informal meaning of the pHL judgment is that, for an initial memory &m satisfying `A`, the probability of running the body of `M.p` in &m and it resulting in termination with a memory &m' satisfying `B`, has the indicated relation to the value of `e`.

Probabilistic Relational Hoare Logic. Variant of HL which reasons about two programs are related to each other. In EasyCrypt, a pRHL judgement has the form

```
equiv [M.p ~ N.q : A ==> B]
```

where `p` is a procedure of module `M` and `q` is a procedure of module `N`; `A` is an assertion (precondition) on memories &m1 and &m2 (where module `M` and `N` will be run, respectively) that may involve

global variables of declared modules as well as parameters of `M.p` and `N.q` which must be respectively interpreted in memories &m1 and &m2; `B` is an assertion (postcondition) on the final memory distributions (since both programs might be probabilistic) which may involve the term `res{1}` which is the returned value of `M.p` and `res{2}` which is the returned value of `N.q`, as well as global variables of declared modules.

The informal meaning of the pRHL judgment is that, for an initial memories &m1 and &m2 satisfying `A`, then the distributions on memories d&m1' and d&m2', respectively obtained by running `M.p` on &m1 and `N.q` on &m2, satisfy `B`.

All of these judgements can be used to write `lemmas` in EasyCrypt, that must be resolved using *tactics*, which are logical rules embodying general reasoning principles, which transform `lemmas` into zero or more subgoals – sufficient conditions for the `lemma` to hold. If there are no more subgoals to solve, we have proved our `lemma`. The following code shows an example of a simple lemma written in EasyCrypt, which formalizes the symmetry property of the equality relation between integers:

```
lemma Sym (x y : int) : x = y => y = x.
```

2.2.2 Jasmin

We use Jasmin to have a concrete implementation of the reference implementation that can be linked with a PM. Jasmin [2] is a framework for developing high-speed and high-assurance cryptographic software, which is structured around the Jasmin programming language and its compiler. The programming language combines high-level (structured control-flow, variables, etc.) and low-level (assembly instructions, flag manipulation, etc.) constructs while guaranteeing verifiability of memory safety and constant-time security. The compiler transforms Jasmin programs into assembly, while preserving behavior, safety, and constant-time security of the source code. Almeida *et al* [2] mentioned that this programming language, while being easily verifiable, is as efficient as some state-of-the-art high-speed software such as qhasm [6].

Programs that are compiled from Jasmin are guaranteed to preserve their behaviour when compiled to assembly, which ensures that properties of programs in Jasmin provably carry to their assembly implementations. Jasmin's compiler has this property proven in Coq[1].

2.2.3 Why EasyCrypt with Jasmin

As previously mentioned, EasyCrypt was chosen because it provides the essential logics to reason about cryptographic constructions using the game-based approach, which is necessary to prove the security property of the generator.

Regarding Jasmin, it was chosen because it allows us to build a low-level efficient algorithm, with the necessary high-level constructs that help us reason about our implementation.

Also, the Jasmin framework provides a way of automatically generating the model of a Jasmin program to EasyCrypt, while having the assurance that we are reasoning about that exact Jasmin code. Moreover, these frameworks when used together have shown good results on building and verifying high-assurance and high-speed cryptographic code, such as the ChaCha20 and Poly1305 implementations proposed by Almeida *et al* [3].

3. RPG Implementation

In this thesis, based on the information obtained from our survey presented in Section 2, we propose a solution for a formal verified RPG, which is first specified in EasyCrypt as a reference implementation, and then implemented in Jasmin.

3.1. Reference Implementation

We propose a reference implementation for an RPG which offers the following policy adjustments: (1) the user can define the password length in the interval $[1; 200]$; (2) the user can choose which sets to use (from Lowercase Letters, Uppercase Letters, Numbers, and Special Characters); (3) the user can define the minimum and maximum occurrences of characters per set.

3.1.1 Definitions on Password Composition Policies

We say that a policy is a record with the following fields: *Length*, *LowercaseMin*, *LowercaseMax*, *UppercaseMin*, *UppercaseMax*, *NumbersMin*, *NumbersMax*, *SpecialMin*, and *SpecialMax*.

Password composition policies can *satisfiable* or *unsatisfiable*. A *satisfiable* password composition policy implies the existence of a password that is able to satisfy all of the different constraints specified in the policy. On the other hand, no password is able to satisfy all of the constraints of an *unsatisfiable* policy.

In our case, it is easy to see that a policy is satisfiable if its field *length* is greater than 0, if all *min* values are non-negative, if all *max* values are greater or equal to the corresponding *min* value, if the sum of all *min* values is less or equal to *length*, and if the sum of all *max* values is greater or equal to *length*. If any of these conditions is not true, then no password is able to satisfy the policy.

We also add to this definition the restriction of the *length* and also all *max* fields being less or equal to

[1] `https://coq.inria.fr/`

200, because we impose that the user cannot generate passwords with a length greater than 200.

3.1.2 Algorithm

The entry point of our algorithm is GENERATEPASSWORD, which receives as input a password composition $policy$ and, if it is satisfiable, a password is generated and returned. Otherwise, a password is not generated and *null* is returned.

Then, to generate passwords, the algorithm follows the idea from the general Algorithm 1 presented in Section 2. Both the random character generator and the string permutation method are very essentially the same as the ones from Algorithm 1.

However, the RNG we decided to implement has some modifications when compared to the ones shown in Section 2.1.5, in order to fix the previously mentioned problem of not having an ideal RNG regarding efficiency.

Regarding the random bytes generator, we use the x86 instruction RDRAND. In this work we assume that this procedure is secure (i.e., samples bytes according to a uniform distribution).

All of these methods are captured by the pseudo-code presented in Algorithm 3.

Algorithm 3 Reference Implementation Pseudo-Code

procedure GENERATEPASSWORD($policy$)
  if $policy$ is $satisfiable$ **then**
    $password \leftarrow \varepsilon$
    foreach $set \in policy.charSets$ **do**
      $set.available \leftarrow set.max$
      for $i = 0, 1, \ldots, set.minOccurrences$ **do**
        $char \leftarrow$ RANDOMCHARGENERATOR(set)
        $password \leftarrow password||char$
        $set.available \leftarrow set.available - 1$
      end for
    end for
    while $len(password) < policy.length$ **do**
      $unionSet \leftarrow \bigcup_{set \in policy.charSets} set$ such that $set.available > 0$
      $char \leftarrow$ RANDOMCHARGENERATOR($unionSet$)
      $password \leftarrow password||char$
      $set.available \leftarrow set.available - 1$
    end while
    $password \leftarrow$ PERMUTATION($password$)
    return $password$
  else
    return $null$
  end if
end procedure

3.2. Reference Implementation in EasyCrypt

In EasyCrypt we call our reference implementation `RPGRef` and we say that it is of type `RPG_T`. Modules of this type must implement a `generate_password` method which receives a `policy` and outputs a `passwrord option`

Regarding type definitions on EasyCrypt, instances of type `char` are integers (which can be directly mapped to the corresponding ASCII character), and both the types `password` and `charSet` are lists of `chars`. The type `policy` is a record type with its fields instantiated as integers.

We note the use of `password option` for the output of the `generate_password` method, which extends the `password` type with the extra element `None`. This allows us to return `Some password` in case the policy is satisfiable, and `None` otherwise.

Regarding methods, it is fairly easy to see how the abstract version of this module maps to the EasyCrypt implementation.

However, there are some small differences. The most noticeable is the definition of union of sets. In the abstract implementation, we simply the standard mathematical notation for union set. In EasyCrypt we have the method `define_union_set` which implements this process. This algorithm can be found online[2].

3.3. Jasmin Implementation

The code of our Jasmin implementation is closely related to the reference implementation. But, being Jasmin a low-level programming language, there are obviously some specific implementation particularities.

Due to the nature of the programming language, the first difference is that everything is represented as bit words. The different policy fields are 64 bit words and characters are 8 bit words. Policies, characters sets, and passwords are arrays of these bit words.

Another difference is on the input/output. The `generate_password` method receives as input two pointers, one where the input policy is stored in memory, and one for the memory region where the program is supposed to write the generated password, in case the policy is satisfiable.

The Jasmin implementation outputs an integer which gives feedback about the generated password. In case the output is positive, it means that the input policy was satisfiable and the Jasmin implementation wrote on memory the generated password. In case it is negative, it means that the input policy was unsatisfiable and nothing was written on memory. The Jasmin implementation also provides some additional feedback on why the policy is unsatisfiable, compared to the reference implementation. The output can take different negative values depending on the reason why the policy unsatisfiable (e.g., if the output is -1 it means that the *length* field is greater than 200).

Another noticeable difference is that arrays must have a constant size in Jasmin. So our variable that

[2] `https://github.com/passcert-project/random-password-generator/blob/main/EC/passCertRPGSpec.ec`

Correctness$^{RPG}(policy)$

if $policy$ is $satisfiable$
  $pwd \leftarrow$ *RPG*.generate_password($policy$)
  return satisfiesPolicy($policy$, pwd)
else
  return isNull(pwd)
fi

Listing 1: Correctness Experiment (Abstract)

stores the union of sets as a size of 76 positions, independently of the sets that compose it. We fill the unnecessary positions with zeros.

The Jasmin implementation is available online[3]. We also provide an example of a simple program calling our Jasmin implementation written in C[4].

3.4. Jasmin Implementation modelled in EasyCrypt

The Jasmin framework implements an embedding of Jasmin programs in EasyCrypt, where x86 instructions over 64-bit words and the memory model of Jasmin are encoded.

This feature enables automatic extraction of Jasmin programs into an equivalent EasyCrypt model of that same program. This is true provided that the Jasmin program is safe [3]. Broadly speaking, safety entails termination, array accesses are in bounds, memory accesses are valid and arithmetic operations are applied to valid arguments.

Given that our RNG is not guaranteed to terminate, one cannot say that the implementation is safe. So we cannot automatically derive that the real Jasmin implementation and the one modelled in EasyCrypt are equivalent. But we argue that, by inspection, these two modules are equivalent.

The code of this EasyCrypt model can be found online[5].

4. Formal Proofs

4.1. Functional Correctness

We say that an RPG is functionally correct if, given any policy, every password it generates satisfies that policy. This property guarantees that users will always get an output according to their expectations.

We follow the standard approach of expressing correctness of a scheme by using a probabilistic experiment that checks if the specification is fulfilled.

Figure 1 shows the *Correctness* experiment, which is parameterized by an RPG implementation that, for any policy, outputs *true* if the RPG behaves according to the specification.

Specifically, if the experiment receives as input a *policy* that is satisfiable, it checks if the parameterized RPG generates a password that satisfies that *policy*. Otherwise, it checks if the RPG outputs *null*.

We want to prove that this experiment parameterized by our reference implementation outputs *true* with probability 1 (for any input *policy*).

4.1.1 EasyCrypt Definition

In EasyCrypt, the correctness experiment is modelled as the module `Correctness`[6], shown in Figure 2.

```
module Correctness(RPG : RPG_T) = {
  proc main(policy:policy) : bool = {
    var pw : password option;
    var satisfied : bool;

    pw <@ RPG.generate_password(policy);
    if(satisfiablePolicy policy) {
      satisfied <- is_some pw /\
                   satisfiesPolicy policy (oget pw);
    }
    else {
      satisfied <- is_none pw;
    }

    return satisfied;
   }
}.
```

Listing 2: Correctness Experiment (EasyCrypt)

The correctness property can be expressed in EasyCrypt as follows:

```
lemma rpg_correctness :
  Pr[Correctness(RPGRef).main : true ==> res] = 1%r.
```

It states that, running the correctness experiment (`main` method) of the `Correctness` module instantiated with our RPG reference implementation, produces the output *true* with probability 1 (for any input *policy*).

4.1.2 EasyCrypt Proof

The proof of this lemma amounts essentially to proving termination of the `main` method, which also entails termination of the generation procedure, while also proving, using Hoare logic, that this method always returns *true*, independently on the policy given as input. These two properties can be expressed by the two following lemmas, respectively:

[3] https://github.com/passcert-project/random-password-generator/blob/main/Jasmin/passCertRPG.jazz

[4] https://github.com/passcert-project/random-password-generator/blob/main/C/RPG_app.c

[5] https://github.com/passcert-project/random-password-generator/blob/main/EC/passCertRPG_jazz.ec

[6] https://github.com/passcert-project/random-password-generator/blob/main/EC/RPGTh.eca

```
lemma c_lossless :
    islossless Correctness(RPGRef).main.

lemma c_correct p :
    hoare [Correctness(RPGRef).main : policy = p
    ==> res].
```

The assertion `islossless` states that `Correctness(RPGRef).main` always terminates, for any input. This can be easily proved if we prove termination of our RPG, since the `main` method is composed by an *if-else* statement, regular assignments, and a call to `RPGRef.generate_password`.

Proving that the RPG terminates amounts to proving that each of the *while* loops of the algorithm will eventually finish. We proved it for all of the *while*s in the algorithm, except for the one from the RNG. In fact, we are not able to prove that this loop terminates, but, as shown in Section 2.1.5, the number of iterations we need to perform before exiting the loop is, most likely, very small, and it will eventually finish. So, we have the axiom

```
axiom rng_ll : islossless RPGRef.rng.
```

which states that the RNG terminates. So, we are able to prove termination of the RPG, thus proving termination of the `Correctness` module.

The second lemma is an Hoare triple, as presented in Section 2.2.1. To prove this Hoare triple, we need to prove that the `main` method outputs a password that satisfies the input policy, in case it is satisfiable, and `None` if it is not satisfiable.

With these two lemmas proved, we can combine them to prove our main lemma `rpg_correctness`, which ensures that our RPG implementation is correct.

4.2. Security

The security of an RPG shall be assessed by measuring how difficult it is, for an attacker, to successfully guess the generated password. Considering that the attacker has no access to side-channels and is only able to see the input and the output of the generation process, the hardest way for the attacker to guess the output is if all the passwords that can possibly be generated by the input policy have the same probability of being generated. This means that, ideally, our RPG should produce a uniform distribution over its output.

So, we say that an RPG is secure if, given any policy, the generated password has the same probability of being generated as any other possible password that satisfies that policy. To prove this property we can use the game-based approach for cryptographic security proofs [18].

We create a module called *IdealRPG* which, in case it receives as input a satisfiable policy, outputs a password sampled from the subset of passwords that satisfy the policy, according to a uniform distribution over that subset.

If the policy is not satisfiable, it outputs *null*. This module is shown in Figure 3. In order to consider our implementation secure, we must show that any program (e.g., attacker) that has oracle access to the *IdealRPG* and our RPG can not distinguish whether it is interacting with one or the other.

proc IdealRPG(policy)

if $policy$ is $satisfiable$
  $password \leftarrow\$ \ p \subset P$
  return $password$
else
  return *null*
fi

Listing 3: Ideal RPG. p is the subset of the set of all possible passwords P that satisfy the given policy.

To achieve this, we use probabilistic relational Hoare Logic (pRHL) to show that both modules' `generate_password` methods produce the same result (they have the same distribution over their output, given any input). We can avoid directly reasoning about the indistinguishability between these two modules, since their implementations are significantly different. By using the game-based approach, we implement intermediate modules that are more closely related, thus breaking the proof into smaller steps, easier to justify.

4.2.1 EasyCrypt Definition

In EasyCrypt we can write the lemma that we need to prove to consider our RPG secure:

```
lemma rpg_security :
  equiv [IdealRPG.generate_password ~
         RPGRef.generate_password :
         ={policy} ==> ={res}].
```

This is a pRHL judgement which means that for all memories &m1 and &m2 (sets of variables of IdealRPG and RPGRef, respectively) if ={*policy*} holds (the input *policy*, has the same value in both memories), then the distribution on memories d&m1' and d&m2', obtained by running the respective methods from the initial memory, satisfy ={*res*} (*res*, the output value, has the same mass in both distributions). If we prove this lemma for our RPG reference implementation, we prove that these methods produce the same distributions over their output, hence establishing security of the RPG reference implementation.

4.2.2 Proof Sketch

To prove the security lemma stated above, we need to establish that the induced distribution from the

execution of RPGRef.generate_password is uniform among all passwords satisfying the policy. It requires fairly detailed reasoning on the distribution level in EasyCrypt. The general structure of the argument follows the structure of Algorithm 1: (1) It starts by generating a password that satisfies the length and the different set bounds defined in the policy, placing them at specific positions; (2) It randomly shuffles the password. The result follows from arguing that the combination of characters after step (1) are sampled according to a uniform distribution, and that the final shuffle allows to reach, with the same probability, any possible password satisfying the policy.

4.3. Jasmin Implementation Verification

4.3.1 Equivalence between Jasmin and Reference Implementation

To prove this equivalence, one needs to prove the equivalence between the corresponding sub-procedures of each module, so we can prove that the main generate_password methods are equivalent. Since in the reference implementation types are modelled using integers, while in Jasmin everything is modelled as 64-bit or 8-bit words, during the proofs one must take into account some upper and lower bounds on arithmetic operations on integers that are represented as bit words, to avoid over and underflows.

In order to make the lemma more readable and easy to reason about, we implement a module of type RPG_T which is parameterized by our Jasmin implementation that serves as a "bridge" between the reference implementation world and the Jasmin world, named ConcreteScheme. This module just transforms policies specified in EasyCrypt and writes them in memory and, after the RPG generates a password, extracts it from memory and transforms it into the type *password*[7].

So, we can easily write the lemma that states that both modules are equivalent, using probabilistic relational Hoare logic (pRHL):

```
lemma implementation_reference_equiv :
  equiv [ConcreteScheme.generate_password ~
         RPGRef.generate_password :
         ={policy} /\ policyFitsW64 policy{2}
         ==> ={res}]
```

It states that running the ConcreteScheme and the reference implementation with the same input *policy*, with the assumption that each field of the policy fits in 64 bits, results in the same output in both programs (the distribution on the possible values for res is the same).

To prove this lemma, we need to prove that if the the policy is satisfiable, the respective distributions on the possible passwords that can be generated are the same. On the other hand, if the policy is unsatisfiable, one must prove that both programs output None.

The second part of the proof is simple. The reference implementation always outputs None when it receives an unsatisfiable policy (as shown in Section 4.1.2), and the ConcreteScheme outputs None if the Jasmin implementation outputs a negative value. So, one needs to check if indeed the Jasmin outputs a negative value whenever it receives an unsatisfiable true, which is easy to verify.

The challenge arises when proving, for satisfiable policies, that the distribution of the Jasmin implementation output mapped into a password is the same as the reference implementation. To prove this property, operands that express equality between the types from one module to the other were naturally defined as such:

```
op EqWordChar word char = W8.to_uint word = char.
op EqWordInt word int = W64.to_uint word = int.
op EqWordIntSet (memSet:W8.t Array76.t)
                (specSet:charSet) =
  forall n, n \in range 0 (size specSet)
  => EqWordChar memSet.[n] (nth (-1) specSet n).
```

The first two operands express equality between bit words and integers (to_uint converts words to integers), and EqWordIntSet expresses equality between word arrays from the Jasmin model and character sets written in EasyCrypt.

Using these simple definitions, we write the lemmas that formalize the equivalence between the corresponding sub-procedures of each module[8].

After proving all these equivalences, it is easy to prove that on satisfiable policies, both modules produce the same output. So, we can prove the lemma that states equivalence between the Jasmin RPG and the reference implementation, implementation_reference_equiv.

4.3.2 Correctness and Security of the Jasmin Implementation

To prove that our Jasmin implementation is functionally correct and secure, we subject it to the correctness experiment and the security game.

Having the equivalence between the reference implementation and the ConcreteScheme module proved and having proved that the reference implementation satisfies these two properties, these two lemmas are trivial to prove. The argument is that we can replace the ConcreteScheme by the reference implementation, since they are interchange-

[7] https://github.com/passcert-project/random-password-generator/blob/main/EC/passCertRPG_jazz_proof.ec

[8] https://github.com/passcert-project/random-password-generator/blob/main/EC/passCertRPG_jazz_proof.ec

able, and then we apply the lemmas where the functional correctness and security of the reference implementation are proved.

5. Evaluation

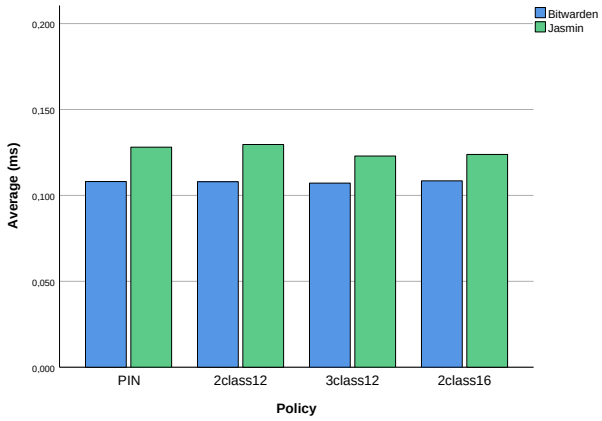
5.1. Integration with Bitwarden

As proof-of-concept, we integrate our solution with the CLI application[9]. It is written in TypeScript so, in order to adapt it to run our code, we use NodeJS *child-process* module to spawn a shell which executes the command that runs our implementation.

5.2. Performance

We evaluate the performance of our implementation by comparing Bitwarden CLI running with its original RPG to it running with out RPG. To do so, we measure the time that it takes to perform the generation of a password for both solutions for 4 different policies.

Results Figure 4 shows the average results.



Listing 4: Comparison of Bitwarden CLI using its own RPG and Bitwarden CLI using our RPG implemented in Jasmin.

The results show that the Bitwarden original RPG is slightly more efficient compared to ours. However, if we think in absolute values, the generation process of our implementation takes essentially the same time, mostly because we are comparing times that are of a really small order of magnitude. Also, one must take into account the overhead of binding our Jasmin implementation with the TypeScript of Bitwarden, which suggests that, without this overhead, our implementation would be even more similar to Bitwarden's.

It is also possible to see that, the complexity of the policies did not have much impact on the performance, at least for these policies which do not sample very big passwords, but are some of the most commonly used.

[9] `https://github.com/bitwarden/cli`

6. Conclusions

We propose a reference implementation of an RPG that, given a password composition policy, is assured to generate passwords compliant with the policy, and we formalize the property that the generator samples the set of passwords according to a uniform distribution. In addition, we realized this reference implementation with a concrete one written in Jasmin that preserves the properties.

As for future work, the next natural step to be done after this work, is to prove with EasyCrypt the security property for the reference implementation, following the proposed structure on Section 4.2.2. Another relevant property that could be proved about our RPG is side-channel protection. The Jasmin framework provides a method that supports these proofs (sometimes automatically) which can also be explored.

During the development of this thesis, two articles were written [11, 10].

Acknowledgments. This work was partially funded by the PassCert project, a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019.

References

[1] N. Alkaldi and K. Renaud. Why do people adopt, or reject, smartphone password managers? 2016.

[2] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017.

[3] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020.

[4] S. Aurigemma, T. Mattson, and L. Leonard. So much promise, so little use: What is stopping home end-users from using password manager applications? 2017.

[5] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub. Easycrypt: A tutorial. In *Foundations of security analysis and design vii*, pages 146–166. Springer, 2013.

[6] D. Berstein. *Writing high-speed software*.

[7] J. F. Ferreira, S. Johnson, A. Mendes, and P. J. Brooke. Certified password quality—a case study using Coq and Linux pluggable authentication modules. In *International Conference on Integrated Formal Methods*, pages 407–421. Springer, 2017.

[8] R. A. Fisher, F. Yates, et al. *Statistical tables for biological, agricultural and medical research, edited by ra fisher and f. yates*. Edinburgh: Oliver and Boyd, 1963.

[9] D. Florencio and C. Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*, pages 657–666, 2007.

[10] M. Grilo, J. Campos, J. F. Ferreira, J. B. Almeida, and A. Mendes. Verified password generation from password composition policies, 2021. Submitted for publication. Draft available from authors.

[11] M. Grilo, J. F. Ferreira, and J. B. Almeida. Towards formal verification of password generation algorithms used in password managers. *arXiv preprint arXiv:2106.03626*, 2021.

[12] T. Hunt. Password reuse, credential stuffing and another billion records in have i been pwned. *troyhunt. com*, 2017.

[13] T. Hunt. Passwords evolved: Authentication guidance for the modern era. troyhunt. com, 2017.

[14] S. Johnson, J. F. Ferreira, A. Mendes, and J. Cordry. Skeptic: Automatic, justified and privacy-preserving password composition policy selection. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 101–115, 2020.

[15] P.-H. Kamp, P. Godefroid, M. Levin, D. Molnar, P. McKenzie, R. Stapleton-Gray, B. Woodcock, and G. Neville-Neil. Linkedin password leak: salt their hide. *ACM Queue*, 10(6):20, 2012.

[16] S. Pearman, S. A. Zhang, L. Bauer, N. Christin, and L. F. Cranor. Why people (don't) use password managers effectively. In *SOUPS*, pages 319–338, 2019.

[17] D. Pereira, J. F. Ferreira, and A. Mendes. Evaluating the accuracy of password strength meters using off-the-shelf guessing attacks. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 237–242. IEEE, 2020.

[18] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch.*, 2004:332, 2004.

[19] S. K. Sood, A. K. Sarje, and K. Singh. Cryptanalysis of password authentication schemes: Current status and key issues. In *2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS)*, pages 1–7. IEEE, 2009.

[20] C. Zuo, Z. Lin, and Y. Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310. IEEE, 2019.