TÉCNICO LISBOA

Formal Verification of Password Generation Algorithms used in Password Managers

Miguel Proença Tavares Ferreira Grilo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. José Carlos Bacelar Almeida

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Manuel Bernardo Martins Barbosa

November 2021

Acknowledgments

Throughout this thesis I have had the fundamental help and guidance from many people. Without them you would not be reading this section, nor any other section, because I would not be capable of doing this work all by myself.

I first want to thank my two supervisors Prof. João F. Ferreira and Prof. José Bacelar Almeida for giving me this opportunity. It was an honour to work with them. They provided me all the technical and academic support that I needed, and whenever I needed. For this reason, I always felt that I could count on them as soon as any doubt emerged in my mind, and that made my job much simpler. I also want to thank all of my Professors and colleagues from the PassCert project, for all the help they provided me.

To my parents that always believed in me. Thank you for all the support, and for always giving your best for my personal and academic development. I could not have made it without you.

Then my sister. Sorry for all the dishes that I did not wash, and the clothes I did not take care of. But you also did those things so I guess we are even. Thanks for sharing this academic years with me. I always felt that you understood the moments I was stressed, and you were always ready to support me. And of course I also want to thank the rest of my family. My grandparents, my uncles and aunts, and my cousins. Thanks for being a great supportive family, as always.

Finally I want to thank my friends. Friends from my hometown Sines and friends I made during university. Thank you all. Thank you for the amazing moments we shared, the inspiring conversations we had, the music we heard, the tennis matches we played (and paddle or whatever it is called), and all the support that you gave me. I feel blessed for having you in my life. And of course you guys from Lisbon know how important you were for my academic success. I would not surely be writing this thesis without you, at least not this early in my lifetime.

Again, thanks everyone. I love you all.

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.

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, Security, EasyCrypt, Jasmin.

Resumo

Gestores de Password são importantes ferramentas que nos permitem utilizar passwords fortes, libertando-nos do esforço cognitivo de as ter de memorizar. Apesar disto, ainda há muitos utilizadores que não confiam totalmente em gestores de passwords. Neste trabalho, focamo-nos numa *feature* que a maioria dos gestores de passwords oferece, que pode ter impacto na confiança dos mesmos, que é o processo de gerar passwords aleatórias. Pesquisamos quais os algoritmos e protocolos mais utilizados e propomos uma solução para uma implementação de referência de um algoritmo de geração de passwords formalmente verificado. Finalmente, concretizamos a implementação de referência em Jasmin, e provamos que esta implementação preserva as propriedades verificadas. Utilizamos EasyCrypt como o ambiente de prova e Jasmin como linguagem de programação.

Palavras Chave

Gestor de Passwords, Geração de Passwords Aleatórias, Verificação Formal, Segurança, EasyCrypt, Jasmin.

Contents

1

Introduction

Contents

Passwords are still one of the most commonly-used means of user authentication. Although there are lots of research around password-based authentication [1–5], there are still many problems and vulnerabilities regarding password authentication, for instance: users still use weak passwords that can be guessed [6] and password leaks seem to keep on happening [7, 8]. So, in general, security experts recommend using password managers for both storing and generating strong random passwords [9]. Despite these recommendations, users tend to reject password managers, mostly because they do not fully trust these applications, or because they are not convenient [10–12].

While trying to understand why users adopt password managers, different researchers observed that users consider the generation of random passwords as one important feature that makes them use these applications [10]. Moreover, users who do not use password managers with built-in password generators more often have weaker passwords and tend to reuse them [11]. Security experts have already warned about how dangerous it is to have weak passwords and, even worse, to reuse those weak passwords [13]. These studies suggest that a strong password generator that users can fully trust is a must-have feature for password managers, 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 [14]. As for this work, we want to prove two properties about the reference implementation: its functional correctness and security. These properties can be described in natural language as such: (1) Functional Correctness = generated passwords always satisfy the input password composition policy; (2) Security = the probability of one password being generated is the same as any other password (given the set of passwords that can be generated according to the password composition policy).

This reference implementation and its properties will be specified in EasyCrypt [15], an interactive framework for verifying the security of cryptographic constructions and protocols. This allows us to reason about the reference implementation directly in the framework.

To understand the state of the art on random password generation, and how we should reason about RPGs, we survey currently used and popular password managers' RPGs.

Regarding the Jasmin implementation, we want to integrate it into a password manager, as a proof-of-concept of our solution. We want to prove the equivalence between this Jasmin implementation and the reference implementation in EasyCrypt, in order to guarantee that our concrete implementation also satisfies the two previously mentioned properties.

Finally, we want to evaluate the efficiency of the Jasmin implementation.

1.2 Contributions

In summary, our contributions are:

- A study of the RPGs of currently used and popular password managers;
- A reference implementation of an RPG modelled in pseudo-code and EasyCrypt;
- Functional Correctness proof of the reference implementation in EasyCrypt;
- Formalization of the security property in EasyCrypt alongside with a short proof sketch;
- Implementation of the RPG in Jasmin;
- Proof of equivalence between the reference implementation and the Jasmin implementation in EasyCrypt, from where one can derive that the Jasmin implementation preserves the desired properties.

This work was developed on the context of the PassCert project[1]. PassCert is a project that tries to combine formal verification of software with password security. Specifically, the objective is to build a proof-of-concept password manager that, through the use of formal verification, is guaranteed to satisfy properties on data storage and password generation, with the final goal of understanding how these proofs can help users gain more trust regarding these applications and increase their adoption. This thesis is the result of our contribution on this project, where we focused on the random generation of passwords, which is a feature that most password managers offer.

Research Papers. Parts of the work presented in this thesis were used in the following research papers:

- ***Miguel Grilo**, João F. Ferreira, and José Bacelar Almeida. Verified Password Generation from Password Composition Policies. Presented at INFORUM 2021 (Comunicação). arXiv preprint arXiv:2106.03626. 2021 [16]*
- ***Miguel Grilo**, João Campos, João F. Ferreira, José Bacelar Almeida, and Alexandra Mendes. Verified Password Generation from Password Composition Policies. Submitted for publication. 2021 [17]*

[1]PassCert is a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019

1.3 Organization of the Document

In Section 2 we present the background needed to understand how we designed our RPG and how the proofs are structured. So, we present the necessary notions on password generation, mostly based on a survey we made to currently used password managers, and we show the logics of EasyCrypt and how proofs are performed. In Section 3 we show how we designed our reference implementation and the concrete instance of it programmed in Jasmin. Section 4 presents how we specified the properties of our RPG and how we proved functional correctness, using EasyCrypt. In Section 5, we demonstrate the process of integrating the Jasmin implementation with a password manager (Bitwarden) as a proof-of-concept of our solution. We also make an evaluation of the performance of our RPG implementation. Finally, in Section 6 we describe our main results, the limitations of this work, and what we think that should be done as future work.

2

Background & Related Work

Contents

In this chapter we present the state of the art on RPGs used in password managers based on a survey to currently used password managers. We also present the technologies that we are going to use for our solution and explain why we have chosen them.

2.1 Random Password Generators Survey

As our first objective, we surveyed the algorithms and protocols for generating random passwords used by different password managers. We must mention that while some password managers only offered the option of generating a random password, many others offered as well the option of generating a random passphrase (a phrase of a set of words typically separated by a special character). Here, we only focus on "regular" passwords because it is still the most common way of user authentication and because while passphrases seem to be an enhancement to usability and maybe to security, it is still not clear if they are indeed better than passwords [18].

We studied a total of 15 password managers, but in this work we focus on three of them: Google Chrome's Password Manager (v89.0.4364.1)[1], Bitwarden (v1.47.1)[2], and KeePass (v2.46)[3]. We have chosen these password managers because they are some of the most popular today and also because they are open-source, which allows us to access their source codes and have a deeper understanding of how they generate random passwords.

Note: Password managers are applications for users to generate and store their passwords safely, so typically they expect a user to be interacting with them. However, some password managers have a more "passive" role in storing passwords, like in Chrome's password manager, where the user can not ask arbitrarily to generate a new random password. These passwords are generated the moment a user registers to a new website automatically. So, sometimes the interface that the password manager offers is not intended for the regular user but for the website that the user is registering to. As these interfaces are identical, we only mention "user" as both the user or the website (unless we want to mention something specific to a website interaction).

2.1.1 Password Composition Policy

All studied password managers allow users to define a password composition policy regarding the password to be generated. These policies define the structure of the password, meaning that normally they are used to specify the password length, the different sets of characters that may appear on the generated password, etc. They are the input that is given to the RPG.

[1] https://source.chromium.org/chromium/chromium/src/+/master:components
[2] https://github.com/bitwarden
[3] https://github.com/dlech/KeePass2.x

	Chrome	Bitwarden	KeePass
Password length	1-200	5-128	1-30000
Available sets	Lowercase Letters Uppercase Letters Alphabetic Numbers Special Characters	Lowercase Letters Uppercase Letters Numbers Special Characters	Lowercase Letters Uppercase Letters Numbers Special Characters Brackets Space Minus Underline
Minimum and maximum occurrences of characters per set	Yes	Yes. Can only define minimum	No
Exclude similar characters	Yes {l o I O 0 1}	Yes {l I O 0 1}	Yes {l I O 0 1 |}
Define by hand a character set	No	No	Yes
Define by hand a character set to be excluded	No	No	Yes
Remove duplicates	No	No	Yes

Table 2.1: Available policy adjustments offered. The Alphabetic set in Chrome is the union of Lowercase Letters and Uppercase Letters. Special Characters in Chrome and Bitwarden are the set {- _ . : !} while in KeePass it is {! " # $ % & ' * + , . / : ; = ? @ \ ^ |}. The Brackets set in KeePass is {( ) { } [ ] ¡ ¿}. The Space, Minus, and Underline are the single element sets {␣}, {-}, and {_} respectively.

Password composition policies are used to strict the space of user-created passwords to preclude easily guessed passwords and thus make passwords more difficult for attackers to guess. While generally speaking this is good for the security of the password, this has the problem of reducing usability. Also, sometimes strict policies can generate some easily guessed passwords (e.g., a policy that enforces the use of lowercase letters, uppercase letters, numbers, and special characters may generate the easily guessable password "P@ssw0rd") [19]. In order to help users avoiding these weak passwords to be generated from strict policies, password managers are often equipped with a password strength meter that warns them about the strength of the password (for example, Bitwarden uses zxcvbn [20], KeePass defines their own strength meter, and Chrome does not use any strength meter).

We gathered the different policy adjustments for the generated passwords that our three studied password managers offer, and we present that information in Table 2.1.

2.1.2 Random Password Generation

Almost all password managers 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. Notice that when the user changes the composition policy, the only impact on the generation process is that it changes the amount of sets we can sample from and the characters that belong to those sets (e.g., if we exclude the similar characters we just remove the respective characters from their sets). The only exception is when the user defines the minimum and maximum occurrence of characters per set, which makes the algorithm less trivial. To show how this is achieved, we present the following pseudo-code (Algorithm 1) which is a generalization of the algorithms of the three password managers, being more closely related to Chrome's algorithm.

Algorithm 1 General Password Generation Algorithm

procedure GENERALGENERATE($policy$)
  $password \leftarrow \varepsilon$
  foreach $set \in policy.charSets$ **do**
    for $i = 1, 2, \ldots, set.minOccurences$ **do**
      $char \leftarrow$ Sample uniformly at random a char from set
      $password \leftarrow password \,||char$ ▷ '||' is the append operator
    end for
  end for
  while $len(password) < policy.passwordLength$ **do**
    $char \leftarrow$ Sample uniformly at random a char from the union of sets which have not yet reached their maximum number of occurrences of characters
    $password \leftarrow password \,||char$
  end while
  Generate a random permutation of $password$
  Output $password$
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 *passwordLength* 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 generates a permutation of *password* and outputs it.

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 'llluunl' 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 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. Algorithm 2 describes this process.

Algorithm 2 General Random Char Generator

1: **procedure** GENERALRANDOMCHARGENERATOR(set)
2: $choice \leftarrow$ Random number in the interval $[0; set.size[$
3: $char = set[choice]$
4: Output $char$
5: **end procedure**

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 [21]. The idea is to randomly choose one character from the original string for each position of the new string. This method is described in pseudo-code in Algorithm 3.

Algorithm 3 General String Permutation

1: **procedure** GENERALPERMUTATION($string$)
2: **for** $i = len(string) - 1, \ldots, 0$ **do**
3: $j \leftarrow$ Random number between 0 and i
4: $aux = string[i]$
5: $string[i] = string[j]$
6: $string[j] = aux$
7: **end for**
8: Output $string$
9: **end procedure**

2.1.5 Random Number Generator

As we can see from Sections 2.1.3 and 2.1.4, the RPG needs to have some source of randomness to generate the characters and a permutation of strings of characters. So, the password generator naturally needs to use an implementation of a Random Number Generator (RNG) that generates random numbers within a range of values, which should be sample according to a uniform distribution.

After inspecting the code of the three password managers, we noticed that Chrome and KeePass had an identical RNG, mainly because they generate numbers from 0 to an input *range*. The main idea of these two password managers 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. This is an efficient way to perform this task.

To generate random bytes, the three password managers considered had different approaches. Chrome uses different system calls based on the operating system the password manager 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 (typically integers are represented as 64-bit words, so the maximum would be $2^{64} - 1$). 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 $2^3 - 1 = 7$ when casting the word to an integer) and if the input *range* is 5, then a result of 1 would be twice as likely as a result of 3 or 4 (since 1 mod $5 \equiv 6$ mod $5 \equiv 1$, while only $3 \equiv 3$ mod 5 and $4 \equiv 4$ mod 5, in the interval $[0; 7]$).

So, some of the possible numbers generated by the random bytes generator must be discarded. This is known as rejection sampling, where we sample values and keep rejecting them until we get one that satisfies our needs. While this not guarantees termination, it allows the RNG to sample numbers uniformly, with a very high probability of terminating in very few tries. This is where Chrome and KeePass slightly differ on their implementation, but they both achieve the same final result.

Chrome's solution to this problem is to first find the maximum 64-bit unsigned integer *maxValue* such that, given an input *range*, $maxValue \equiv range - 1$ mod $range$ (i.e., the greatest multiple of $range$ that can be written in 64 bits, subtracted by 1). Then, generate 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.

KeePass's algorithm first generates a random 64-bit unsigned integer *value* and then calculates $value$ mod $range$ (where *range* is again the input). Then, it verifies if the last element of the "block" can be generated. By block we mean a sequence of contiguous integers $i, i + 1, ..., i + range$ such that by

applying mod $range$ to each element of the sequence, we respectively obtain $0, 1, ..., range - 1$. If the last element can be generated this means that our random number is fine and the algorithm outputs it. Otherwise, it keeps on sampling numbers until it samples a satisfactory one.

Chrome and KeePass RNG algorithms are described in pseudo-code in Algorithm 4.

Bitwarden RNG is more complex since it generates numbers from an arbitrary minimum value up to an arbitrary maximum value. Since in our algorithm we only need to generate numbers from 0 to any given maximum value, the simpler implementations of Chrome and KeePass are sufficient for us to focus on.

Algorithm 4 RNGs with maximum range

procedure CHROMERNG($range$)
    $maxValue \leftarrow (uint64.maxValue/range) * range - 1$ ▷ uint64.maxValue is the maximum value an unsigned integer with 64 bits can take; '/' is the Euclidean division
    do
        $value \leftarrow$ (uint64) GenerateRandomBytes
    while $value > maxValue$
    return $value$ mod $range$
end procedure

procedure KEEPASSRNG($range$)
    do
        $genValue \leftarrow$ (uint64) GenerateRandomBytes
        $value \leftarrow genValue$ mod $range$
    while $(genValue - value) > (uint64.maxValue - (range - 1))$
    return $value$
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. We will demonstrate with an example only for Chrome's RNG, because for KeePass it follows the same idea.

For example if, again, we use 3-bit words to represent our integers (maximum value of 7 after casting words to integers), and if the input *range* is 4, Chrome's 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 (in this particular case we would be sure that there would not be any sample rejected).

Of course that for 64-bit words, with a small input *range* (which is the most common case when using an RNG to sample characters from a set) this does not affect much the efficiency of the algorithm, but we still took this into consideration and we will improve this rejection sampling in our RNG.

2.2 Technology Used

In this section we present the chosen frameworks used to implement our solution and why we have chosen them, and also present other frameworks that could have possibly been used.

2.2.1 EasyCrypt

We use EasyCrypt to implement our RPG reference implementation and to reason about it. EasyCrypt [15] 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 ordinary assignments, conditional statements, while loops, procedure calls, return statements, and random sampling operations (assigning values chosen at random from distributions to variables). Figure 2.1 shows an example of a simple module written in EasyCrypt.

```
module M(N : T) = {
  var x : int
  proc init(bnd : int) : unit = {
    x <$ [-bnd .. bnd];
  }
  proc get() : int = {
    return x;
  }
  proc main() : int = {
    return N.main(get());
  }
}.
```

Figure 2.1: Example of a module in EasyCrypt. This module is parameterized by another module `N` of type `T`. In the `init` method, `x` is sampled at random from the uniform distribution over the integers belonging to the interval $[-bnd..bnd]$.

Built around these modules, EasyCrypt implements program logics for proving properties of imperative programs. It has four logics: Hoare Logic (HL), probabilistic Hoare Logic (pHL), probabilistic relational Hoare Logic (pRHL) [22] and an ambient higher-order logic for proving general mathematical facts and connecting judgements in the other logics.

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.
```

This information and more can be obtained from the EasyCrypt reference manual [23].

2.2.2 Jasmin

We use Jasmin to have a concrete implementation of the reference implementation (which is then compiled to assembly) so we can link this module to a password manager. Jasmin [14] 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* [14] mentioned that this programming language, while being easily verifiable, is as efficient as some state-of-the-art high-speed software such as qhasm [24].

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 [25].

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 and verify the properties of 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* [26]. These Jasmin implementations have their functional correctness and constant-time security verified in EasyCrypt, and they outperform the fastest non-verified counterparts. SHA3 was also built and proven correct, secure, and time-attack resistant using this methodology, matching the performance of state-of-the-art implementations and even outperforming some of them [27].

2.2.4 Other Frameworks

There were other interesting options for the programming language and the proof framework besides the the ones we have chosen.

Regarding the programming language, one can obviously construct efficient cryptographic code using a low-level programming language like assembly. The problem with using assembly code is precisely the fact that it is a low-level language which makes it hard to implement such algorithms, and then to understand, analyze, and verify them.

One solution to this problem is to use an higher-level language such as C. However, in order to obtain efficient code, extensive optimizations must be performed which are not achievable even by modern, highly-optimized compilers. Even if a compiler does that, it might still not be the best option because while such optimizations should preserve functionality and correctness they might lead to some security vulnerabilities [28]. HACL* [29] is a formally verified library of cryptographic code written in assembly that uses this method. HACL* takes state-of-the-art optimized C implementations and them adapt them to F* [30]. The security formal proofs are made in the F* code and then these programs are compiled to C and then to assembly. This is a good reason for us to chose Jasmin as our programming language, over an higher-level language like C. Also, the algorithms implemented in Jasmin and verified with EasyCrypt are more efficient than those available in HACL* [26, 27].

Bond *et al* proposed Vale [31], a tool and a programming language for proving functional correctness and side-channel protection of cryptographic constructions. Vale is a flexible framework where implementations are written in annotated assembly, which are then translated into an abstract syntax tree (AST), while also generating proofs about that AST that are verified with an SMT solver. Besides being portable across different architectures, Vale produces really efficient code, yet not as efficient as unverified assembly code.

CertiCrypt [32] is, like EasyCrypt, an automated framework to construct and verify security proofs of cryptographic systems using the game-playing method. It is built on top of Coq [25], which is a more general formal proof framework. So, it provides generality (its language is expressive enough to capture all notions and assumptions used by cryptographers) and high-assurance. Besides these, CertiCrypt mechanizes most of the techniques required for game-based arguments. The main issue with CertiCrypt is that it lacks automation.

3

Random Password Generator Implementation

Contents

In this thesis, based on the information obtained from our survey presented in Section 2, we propose a reference implementation of an RPG. We first present it in pseudo-code and then we specify it in EasyCrypt. Moreover, we implement it using Jasmin, which can then be compiled to assembly. Both the reference implementation and the Jasmin implementation have two properties verified: their functional correctness and security.

Our solution follows the structure depicted in Image 3.1.

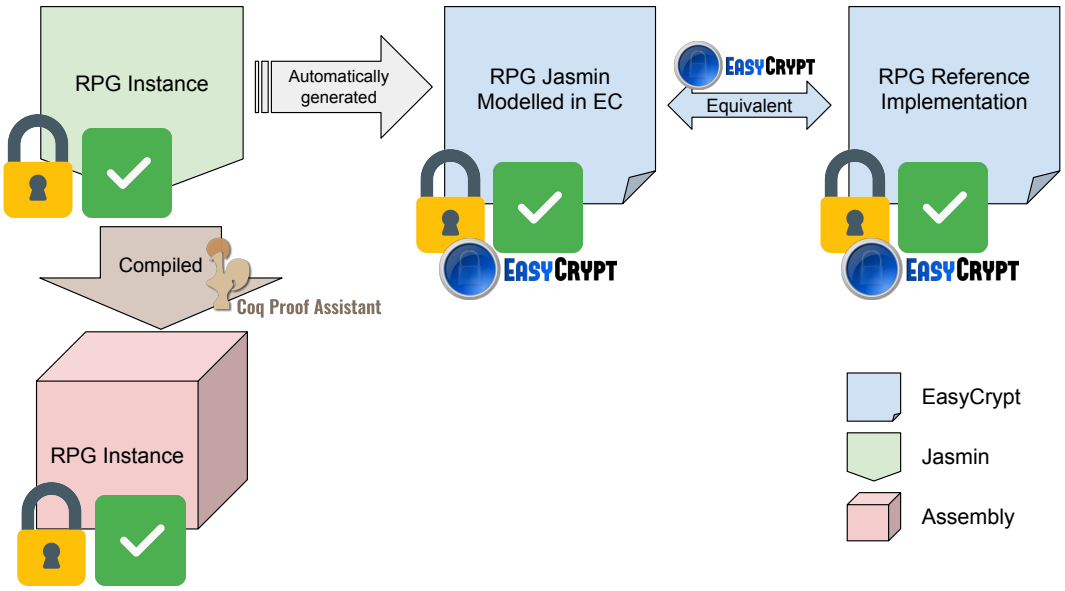

Figure 3.1: Structure of the solution.

Our methodology, based on the approach for building high-assurance and high-efficient cryptographic implementations proposed by Almeida *et al* [26], follows these lines: we first model the reference implementation in EasyCrypt, and we prove the desired properties. Then, we implement it in Jasmin and, using the Jasmin framework, we automatically extract an equivalent version of it modelled in EasyCrypt. We prove the equivalence between this model and the reference implementation, concluding that it also satisfies the properties, from where we can derive that the Jasmin implementation is also correct and secure. Finally, as shown in Section 2.2.2, the assembly version of our implementation also preserves the properties from Jasmin, which is a feature that comes from the Jasmin framework and its formally verified compiler.

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.

First, we write some definitions and present the pseudo-code of our reference implementation, and then we show some concrete definitions, alongside the algorithm, written in EasyCrypt.

3.1.1 Definitions on Password Composition Policies

Taken into consideration the specification presented at the beginning of this section, we say that a policy is a record with the following fields: *Length*, *LowercaseMin*, *LowercaseMax*, *UppercaseMin*, *Uppercase-Max*, *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 200. While there could be, of course, passwords with a length greater than 200, since the user can only define passwords with a length within the interval $[1; 200]$, we also add this as a restriction on our definition. If, for some reason, our algorithm receives as input a composition policy that has its *length* defined to a value greater than 200, we say it is unsatisfiable (we do the same for *max* values greater than 200).

3.1.2 Algorithm

The algorithm follows the idea from the general Algorithm 1 presented in Section 2.

The entry point is the procedure 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.

To generate a random password, the algorithm first randomly generates characters from the sets that have a *min* value greater than 0, and appends them to the *password* (initially an empty string). Then, it randomly generates characters from the union of all sets which have fewer occurrences of characters in *password* than their *max* value defined in the policy until the size of *password* becomes equal to the length defined in the policy. Finally, it generates a random permutation of the string, and returns it.

Both the random character generator and the string permutation method are trivial and follow the idea presented in Algorithm 1.

However, the random number generator 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. The issue of not using all the possible values that would generate a uniform random distribution happens when the maximum integer written in an n-bit word plus 1 is a multiple of the input *range*. Both Chrome's and KeePass's do not check this corner case. In our RNG, before calculating the maximum value that we can accept, we just check if the maximum number written in a 64-bit word plus 1 is a multiple of the input. If it is, we can sample without the need of rejecting samples. As discussed in Section 2.1.5, this algorithm is not guaranteed to terminate. But, since the maximum possible value for the input to this algorithm is 76 (when we are sampling characters from the union of all sets), and since we are representing integers with 64-bit words (which allows us to sample integers in the interval $[0, 2^{64} - 1]$), the probability of a sample being rejected is very small, thus we can say that the algorithm terminates in "expected polynomial time".

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 5.

3.2 Reference Implementation in EasyCrypt

In this section we describe how we have specified in EasyCrypt the pseudo-code presented in Section 3.1.

We call our reference implementation `RPGRef` and we say that it is of type `RPG_T`. Modules of this type implement the following interface

```
module type RPG_T = {
  proc generate_password(policy:policy) : password option
}.
```

where the header of the `generate_password` method is defined. This method should receive a `policy` and output a `password option`.

Figure 3.2 shows the definitions of the necessary types used to reason about password generation. 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 the previously mentioned 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

Algorithm 5 Reference Implementation Pseudo-Code

procedure GENERATEPASSWORD($policy$)
  if $policy$ is $satisfiable$ **then**
    $password \leftarrow \varepsilon$
    foreach $set \in policy.charSets$ **do**
      $set.available \leftarrow set.max$ ▷ field $available$ stores the number of characters we can still sample from set
      for $i = 0, 1, 2, \ldots, set.minOccurrences$ **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

procedure RANDOMCHARGENERATOR(set)
  $choice \leftarrow RNG(set.size)$
  return $set[choice]$
end procedure

procedure PERMUTATION($string$)
  for $i = len(string) - 1, \ldots, 0$ **do**
    $j \leftarrow RNG(i)$
    $aux = string[i]$
    $string[i] = string[j]$
    $string[j] = aux$
  end for
  return $string$
end procedure

procedure RNG($range$)
  $modValue \leftarrow uint64.maxValue$ mod $range$
  if $modValue = range - 1$ **then**
    $maxValue \leftarrow uint64.maxValue$
  else
    $maxValue \leftarrow uint64.maxValue - modValue - 1$
  end if
  do
    $value \leftarrow \#RDRAND$
  while $value > maxValue$
  return $value$ mod $range$
end procedure

the policy is satisfiable, and `None` otherwise.

Regarding the reference implementation 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 say that the the *unionSet* variable takes the union of all sets such that their *max* values are greater than 0, using mathematical notation. In EasyCrypt we have the method `define_union_set` which implements this process. The main idea of this method is similar to the abstract version, where the algorithm creates a union of the sets which have not already reached their maximum number of occurrences, by appending them together.

This algorithm is in Appendix A, but it can also be found online[1].

```
type char = int.
type password = char list.
type charSet = char list.
type policy = {
  length : int;
  lowercaseMin : int;
  lowercaseMax : int;
  uppercaseMin : int;
  uppercaseMax : int;
  numbersMin : int;
  numbersMax : int;
  specialMin : int;
  specialMax : int
}.
```

Figure 3.2: Type definitions.

3.3 Jasmin Implementation

We have implemented in Jasmin our proposed reference implementation, in order to have a concrete module that could be integrated into a password manager.

The code is, naturally, closely related to the reference implementation. But, being Jasmin a low-level programming language, there are obviously some specific implementation details that are different from the more abstract reference implementation written in EasyCrypt.

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: (1) `policy_addr` which is a pointer to a region of memory where the program expects to receive, in contiguous positions, the different fields of the password composition policy; (2) `output_addr` which is a pointer for the memory zone where the program is supposed to write the generated password, in case the policy is satisfiable. The reference implementation simply receives as input an instance of the more abstract type `policy` and outputs a `password option`.

The Jasmin implementation outputs an integer which gives feedback about the generated password.

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

In case the output is positive (i.e., 1), it means that the input policy was satisfiable and the Jasmin implementation wrote on memory (starting on address `output_addr`) 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 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 in Appendix A, and it is also available online[2]. We also provide an example of a simple program calling our Jasmin implementation written in C[3].

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 [26]. 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[4].

[2] https://github.com/passcert-project/random-password-generator/blob/main/Jasmin/passCertRPG.jazz
[3] https://github.com/passcert-project/random-password-generator/blob/main/C/RPG_app.c
[4] https://github.com/passcert-project/random-password-generator/blob/main/EC/passCertRPG_jazz.ec

4

Formal Verification

Contents

In this chapter, we show how we specified our properties in an informal way, followed by the specification in EasyCrypt. Then, we demonstrate the reasoning behind the proofs.

We first show how we proved the two properties in EasyCrypt for the reference implementation, and then we show how, by proving the equivalence between the reference implementation and the Jasmin implementation, we can derive that the Jasmin implementation is also functionally correct and secure.

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 4.1 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*).

Correctness$^{RPG}(policy)$

$pwd \leftarrow$ *RPG*.generate_password($policy$)

if $policy$ is $satisfiable$

  return satisfiesPolicy($policy$, pwd)

else

  return isNull(pwd)

fi

Figure 4.1: Correctness Experiment (Abstract)

4.1.1 EasyCrypt Definition

In EasyCrypt, the correctness experiment is modelled as the module `Correctness`, shown in Figure 4.2. It is parameterized by a password generator implementation (being `RPG_T` its signature), and has a single method `main` encoding the experiment. The experiment simply executes the `RPG.generate_password` method and, depending on the satisfiability of the policy, either checks if the generated password satisfies it, or if it is equal to `None` (`is_some` and `is_none` are predicates that query the constructor used in an

optional value, and `oget` extracts a password from it). This EasyCrypt specification is available online[1].

```
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;
   }
}.
```

Figure 4.2: 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:

```
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`.

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

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. These ideas can be expressed with the following lemmas:

```
lemma rpg_correctness_sat_pcp_hl (p:policy) :
  hoare [RPGRef.generate_password : policy = p /\
          satisfiablePolicy p
          ==>
          is_some res /\ satisfiesPolicy p (oget res)].
```

and

```
lemma rpg_correctness_unsat_pcp_hl (p:policy) :
  hoare [RPGRef.generate_password : policy = p /\
          !(satisfiablePolicy p)
          ==>
          res = None].
```

The second lemma is trivial to prove, because the first thing our RPG implementation does is to check if the input policy is satisfiable. If it is not, our RPG outputs `None`.

To simplify the reasoning around the first property, when the policy is satisfiable, one can separate the proof into two steps: first we prove that the length defined in the policy is satisfied, and then we prove that the different bounds of minimum and maximum occurrences per set are also satisfied. This means that we should first prove the lemmas

```
lemma rpg_correctness_length_hl (p:policy) :
  hoare [RPGRef.generate_password : policy = p /\
          satisfiablePolicy p
          ==>
          is_some res /\ satisfiesLength p (oget res)].
```

and

```
lemma rpg_correctness_bounds_hl (p:policy) :
  hoare [RPGRef.generate_password : policy = p /\
          satisfiablePolicy p
          ==>
          is_some res /\
          satisfiesBounds p (oget res)].
```

Finally, we can combine these two lemmas to prove the lemma `rpg_correctness_sat_pcp_hl` since we can use `hoare[ C : P ==> Q1]` and `hoare[ C : P ==> Q2]`, to conclude `hoare[ C : P ==> Q1 /\ Q2]`. Using `rpg_correctness_sat_pcp_hl` and `rpg_correctness_unsat_pcp_hl`, we can prove the lemma `c_correct` using Hoare logic rules. With the lemmas `c_lossless` and `c_lossless` proved, we can combine them to finally prove our main lemma `rpg_correctness`, which ensures that our RPG implementation is correct.

4.2 Security

The security of a random password generator 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 [33, 34].

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 4.3. 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

Figure 4.3: Ideal RPG. p is the subset of the set of all possible passwords P that satisfy the given policy.

To achieve this, we can 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 can implement intermediate modules that are more closely related, thus breaking the proof into smaller steps that are easier to justify.

4.2.1 EasyCrypt Definition

To construct the *IdealRPG* module, we start by axiomatizing uniform distributions over the type of *passwords*:

```
op dpassword : password distr.
(*1*) axiom dpassword_ll : is_lossless dpassword.
(*2*) axiom dpassword_uni : is_uniform dpassword.
(*3*) axiom dpassword_supp : forall p, p \in dpassword => validPassword p.
```

The operator `dpassword` is the declared distribution over the type `password`. The axioms defined are important properties about this distribution: (1) *lossless* means that it is a proper distribution (its probability mass function sums to one); (2) *uniform* means that all elements in its support have the same mass; (3) the support of the distribution is the set of all valid passwords (`validPassword` checks if passwords have their length belongs in the interval [1,200] and if all their characters belong to one of the four character sets). This distribution can be used to construct the *IdealRPG* module that meets the requirements for our RPG security definition.

```
module IdealRPG = {
  proc generate_password(policy:policy) = {
    var pw;
    var out;
    if(satisfiablePolicy policy) {
      pw <$ dpassword \ doesNotSatisfyPolicy policy;
      out <- Some pw;
    } else {
      out <- None;
    }
    return out;
  }
}.
```

Figure 4.4: Ideal RPG (EasyCrypt)

In this module, a password is sampled if the *policy* is satisfiable, otherwise outputs *None*. The sampling makes use of the axiomatized distribution over passwords, restricting its support by removing

the passwords that do not satisfy the policy. Given these definitions, 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. Here, we present a proof sketch. 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. In the course of the formalisation of the above points, auxiliary results such as the correctness of the well-known probabilistic algorithm of rejection sampling for the RNG, and the Fisher-Yates shuffle algorithm (procedure `Permutation`) have to be tackled.

4.3 Jasmin Implementation Verification

In this section we prove how the Jasmin code modelled in EasyCrypt, as described in Section 3.4, also satisfies the properties of our reference implementation. To do so, we first need to prove that the Jasmin RPG modelled in EasyCrypt and the reference implementation are equivalent, and then we can prove that the Jasmin code also satisfies both the correctness experiment and the security game.

4.3.1 Equivalence between Jasmin and Reference Implementation

As shown in Section 3, these implementations are very identical, so this proof is somewhat straightforward. 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. What make this proof not as trivial as it might seem is the fact that in the reference implementation types are modelled using integers, while in Jasmin everything is modelled as 64-bit or 8-bit words. So, during the proofs one must take into account, for example, 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 decided to 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. We call that module `ConcreteScheme`.

Concrete Scheme. Ideally, this module is the simplest program that makes use of our implementation to generate passwords. Since our Jasmin implementation does not receive as input a policy, `ConcreteScheme` 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*. This module chooses two arbitrary memory positions for the RPG input (the policy address and the outputted password address). One must just be careful to not overlap this positions (i.e., leave enough memory positions to write a policy). In Figure 4.5 we show the `generate_password` method of this module, and the full module is shown in Appendix A and can also be found online[2].

```
proc generate_password(policy:policy) : password option = {
  var policyAddr, pwdAddr, output : W64.t;
  var pwd : password;
  var pwdOpt : password option;

  policyAddr <- W64.of_int 0;
  pwdAddr <- W64.of_int 1000;
  policySpecToMem(policy, Glob.mem, policyAddr);
  output <- M.generate_password(policyAddr, pwdAddr);
  if (output \slt W64.zero) {
    pwdOpt <- None;
  } else {
    pwd <- pwdMemToSpec(policy.`length, Glob.mem, pwdAddr);
    pwdOpt <- Some pwd;
  }
  return pwdOpt;
}
```

Figure 4.5: Main method of `ConcreteScheme` which calls our Jasmin implementation, and makes the necessary transformations from the reference implementation types to the Jasmin model types, and vice-versa.

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

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 words written in 8 or 64 bits 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. In the latter definition we say that for all entries of `memSet` from 0 to the size of `specSet`, the words that is stored in each entry and the characters stored in respective entries on `specSet` are equal. We need to define this equality as such because in Jasmin `union_set` has a constant size of 76, and is filled with zeros in case we are not using it entirely.

Using this simple definitions, we can write the lemmas that formalize the equivalence between the corresponding sub-procedures of each module.

RNG. The property that states that the RNG is equivalent in Jasmin and the reference implementation is formalized with the following lemma:

```
lemma imp_ref_rng_equiv _range :
  equiv[M.rng ~ RPGRef.rng : range{2} = _range /\
                             EqWordInt range{1} range{2} /\
                             0 < to_uint range{1} < W64.modulus
                               ==>
                             EqWordInt res{1} res{2} /\
                             0 <= res{2} < _range].
```

It states that running both RNGs (`M` is our Jasmin RPG) for the same input `range`, assuming it belongs to the interval $]0, 2^{64} - 1]$ (which will always happen when calling the RNGs in our implementations), results in the same output in both programs (the distribution on the possible values for `res` is the same).

Random Char Generator. The property that states that the random char generators are equivalent in Jasmin and the reference implementation is formalized with the following lemma:

```
lemma imp_ref_rcg_equiv :
  equiv[M.random_char_generator ~ RPGRef.random_char_generator :
          EqWordIntSet set{1} set{2} /\
          EqWordInt range{1} (size set{2}) /\
          0 < to_uint range{1} < W64.modulus
           ==>
          EqWordChar res{1} res{2}].
```

It states that running both random character generators for the same input `set`, results in the same output in both programs (the distribution on the possible values for `res` is the same).

Random Char Generator. Finally, the property that states that the methods that define the union of sets are equivalent in Jasmin and the reference implementation is formalized with the following lemma:

```
lemma imp_ref_define_union_set_equiv :
  equiv[M.define_union_set ~ RPGRef.define_union_set :
        EqWordInt lowercase_max{1} nLowercase{2} /\
        EqWordInt uppercase_max{1} nUppercase{2} /\
        EqWordInt numbers_max{1} nNumbers{2} /\
        EqWordInt special_max{1} nSpecial{2} /\
        EqWordIntSet lowercase_set{1} lowercaseSet{2} /\
        EqWordIntSet uppercase_set{1} uppercaseSet{2} /\
        EqWordIntSet numbers_set{1} numbersSet{2} /\
        EqWordIntSet special_set{1} specialSet{2} /\
        size lowercaseSet{2} = 26 /\
        size uppercaseSet{2} = 26 /\
        size numbersSet{2} = 10 /\
        size specialSet{2} = 14
         ==>
        to_uint res{1}.`1 = size res{2} /\
        EqWordIntSet res{1}.`2 res{2}
   ].
```

It states that running both methods that define the union of sets assuming that all four sets are equal and that the available number of characters that we can sample from each set are equal (which will always happen when calling these methods in our implementations), results in the same output in both programs (the distribution on the possible values for `res` is the same).

The proof to these lemmas can be found online[3].

After proving all these equivalences, it is easy to prove that on satisfiable policies, both modules produce the same output. So, we 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 can subject it to the correctness experiment and the security game, respectively. Since this experiment is expected to receive a module of type `RPG_T` which receives inputs of type `policy` and returns values of type `password option`, we can use the `ConcreteScheme`. These properties are formalized with the following lemmas:

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

```
lemma concrete_rpg_correctness &m policy :
  policyFitsW64 policy =>
  Pr[Correctness(ConcreteScheme).main(policy) @ &m : res] = 1%r.
```

and

```
lemma concrete_rpg_security :
  policyFitsW64 policy =>
  equiv [IdealRPG.generate_password ~ ConcreteScheme.generate_password :
  ={policy} ==> ={res}].
```

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

5

Evaluation

Contents

In this chapter we show how our Jasmin implementation can be integrated with one of the studied password managers: Bitwarden. This integration is a proof-of-concept of our solution and it is also used to evaluate the performance of our solution, by comparing the performance of Bitwarden using its original RPG and ours, in order to address if having a formally verified implementation of an RPG comes with a performance penalty.

5.1 Integration with Bitwarden

Bitwarden has different applications where one can run their services: a browser extension, a desktop application for different operating systems, a mobile application and a command line interface (CLI). As proof-of-concept, we integrate our solution with the CLI application[1], because our RPG is written in Jasmin (then compiled to assembly), so it is easier to integrate with an application that runs on a terminal. Moreover, it simplifies the process of running the performance tests.

To integrate, we adapted a library from Bitwarden named *jslib*, which encapsulates all the methods that are shared between the organization's different applications, including the random password generator algorithm. This library is written in TypeScript so, in order to adapt the *jslib* to run our code, we use NodeJS *child-process* module to spawn a shell which executes the command that runs our implementation. With this binding, we are able to successfully run a Bitwarden CLI application that uses our Jasmin implementation when asked to generate a random password.

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 (using the implementation described in the previous section). To do so, we measure the time that it takes to perform the generation of a password for both solutions for 4 different policies. Out of these 4 policies, 3 of them are policies proposed by Shay *et al* [19] that were shown to be secure – *2class12*, *3class12*, and *2class16*. These policies vary on the length (12 and 16) and the number of used classes (2 and 3). We also used a policy named PIN which are the passwords composed of 4 numbers, in order to see if reducing the complexity of the policy had a significant impact on the performance.

Environment All measurements were performed on an Intel i7-8750H processor clocked at 2,20GHz, running Ubuntu 20.04.3 LTS, kernel release 5.11.0-38-generic.

[1] https://github.com/bitwarden/cli

Results Figure 5.1 shows the results. We sampled 200 password for both generators for each policy, and we made an average of the time taken to sample each password, which is presented in milliseconds.

Figure 5.1: 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.

Moreover, this process is not expected to be used many times, compared to, for example, an algorithm of a symmetric cipher that might be used in a protocol multiple times. So, in practice, the results show that our implementation is feasible and that it works as efficiently as a state-of-the-art implementation.

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.

6

Conclusion

Contents

Finally, in this chapter we will present the main conclusions of our work, and discuss some of the limitations of it and the future steps to be done.

6.1 Conclusions

In this work we proposed solution that partially solves the issue of users not trusting password managers, which focuses on the process of generating random passwords.

Keeping in mind the problem to be solved, we propose 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. We proved the equivalence between this module and the reference implementation, which all add up to a formally verified concrete implementation that can be integrated into any password manager.

6.2 System Limitations and 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. This is a commonly proved property for cryptographic constructions, that would make our solution more robust. To prove such property, one has to prove that the RPG is constant-time (i.e., regardless of the input, the time complexity of the algorithm is the same). The effort needed to implement this would essentially amount to make constant lookups to the arrays when getting characters from the sets. The Jasmin framework provides a method that supports these proofs (sometimes automatically) which can also be explored.

We found two major limitations on our work: the first one is the assumption on the underlying random bytes generator. It would be interesting to make the security proof without assuming that the random bytes generator is secure. To do so one needs to prove that the random bytes generator is indistinguishable from the Ideal RPG constructed in Section 4.2 for a certain class of constrained attackers, where one could argue that, in practice, the random bytes generator samples bytes uniformly at random. If one is able to prove side-channel protection and prove security without this assumption, the final result would be a really robust RPG.

The other limitation is that we have fixed the four sets from which characters can be sampled. It would be an interesting feature to extend the generator with the possibility of using arbitrary characters

set, which would allow for more flexibility on generating passwords, thus increasing compliance with different password rules and offering more usability to the user.

Bibliography

[1] C.-L. Lin, H.-M. Sun, and T. Hwang, "Attacks and solutions on strong-password authentication," *IEICE transactions on communications*, vol. 84, no. 9, pp. 2622–2627, 2001.

[2] 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*, 2020, pp. 101–115.

[3] 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*. Springer, 2017, pp. 407–421.

[4] 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)*. IEEE, 2020, pp. 237–242.

[5] 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)*. IEEE, 2009, pp. 1–7.

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

[7] 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)*. IEEE, 2019, pp. 1296–1310.

[8] 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*, vol. 10, no. 6, p. 20, 2012.

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

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

[11] S. Pearman, S. A. Zhang, L. Bauer, N. Christin, and L. F. Cranor, "Why people (don't) use password managers effectively," in *Fifteenth Symposium On Usable Privacy and Security (SOUPS 2019). USENIX Association, Santa Clara, CA*, 2019, pp. 319–338.

[12] 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.

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

[14] 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*, 2017, pp. 1807–1823.

[15] 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*. Springer, 2013, pp. 146–166.

[16] 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.

[17] 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.

[18] M. Keith, B. Shao, and P. Steinbart, "A behavioral analysis of passphrase design and effectiveness," *Journal of the Association for Information Systems*, vol. 10, no. 2, p. 2, 2009.

[19] R. Shay, S. Komanduri, A. L. Durity, P. Huh, M. L. Mazurek, S. M. Segreti, B. Ur, L. Bauer, N. Christin, and L. F. Cranor, "Designing password policies for strength and usability," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 4, pp. 1–34, 2016.

[20] D. L. Wheeler, "zxcvbn: Low-budget password strength estimation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 157–173.

[21] 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.

[22] G. Barthe, B. Grégoire, and S. Z. Béguelin, "Probabilistic relational hoare logics for computer-aided security proofs," in *International Conference on Mathematics of Program Construction*. Springer, 2012, pp. 1–6.

[23] "Easycrypt reference manual." [Online]. Available: https://www.easycrypt.info/documentation/refman.pdf

[24] D. Berstein, *Writing high-speed software*. [Online]. Available: https://cr.yp.to/qhasm.html

[25] "The coq proof assistant." [Online]. Available: https://coq.inria.fr/

[26] 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)*. IEEE, 2020, pp. 965–982.

[27] J. B. Almeida, C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton, and P.-Y. Strub, "Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of sha-3," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1607–1622.

[28] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Security and Privacy Workshops*. IEEE, 2015, pp. 73–87.

[29] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[30] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss *et al.*, "Dependent types and multi-monadic effects in f," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270.

[31] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 917–934.

[32] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 90–101.

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

[34] M. Bellare and P. Rogaway, "Code-based game-playing proofs and the security of triple encryption." *IACR Cryptol. ePrint Arch.*, vol. 2004, p. 331, 2004.

A

Reference Implementation

Reference Implementation written in EasyCrypt:

```
proc generate_password(policy:policy) : password option = {
  var outputPassword : password option;
  var generatedPassword : password;
  var unionSet : charSet;
  var randomChar : char;
  var i : int;
  var lowercaseAvailable, uppercaseAvailable, numbersAvailable, specialAvailable : int;
  (* if policy is satisfiable, return Some password *)
  if(satisfiablePolicy policy) {
    lowercaseSet <@ get_lowercaseSet();
    uppercaseSet <@ get_uppercaseSet();
    numbersSet <@ get_numbersSet();
    specialSet <@ get_specialSet();
    generatedPassword <- [];
    lowercaseAvailable <- policy.`lowercaseMax;
    uppercaseAvailable <- policy.`uppercaseMax;
    numbersAvailable <- policy.`numbersMax;
```

```
specialAvailable <- policy.`specialMax;
(* generate characters with min values defined *)
if (0 < lowercaseAvailable) {
  i <- 0;
  while (i < policy.`lowercaseMin) {
    lowercaseAvailable <- lowercaseAvailable - 1;
    randomChar <@ random_char_generator(lowercaseSet);
    generatedPassword <- generatedPassword ++ [randomChar];
    i <- i + 1;
  }
}
if (0 < uppercaseAvailable) {
  i <- 0;
  while (i < policy.`uppercaseMin) {
    uppercaseAvailable <- uppercaseAvailable - 1;
    randomChar <@ random_char_generator(uppercaseSet);
    generatedPassword <- generatedPassword ++ [randomChar];
    i <- i + 1;
  }
}
if (0 < numbersAvailable) {
  i <- 0;
  while (i < policy.`numbersMin) {
    numbersAvailable <- numbersAvailable - 1;
    randomChar <@ random_char_generator(numbersSet);
    generatedPassword <- generatedPassword ++ [randomChar];
    i <- i + 1;
  }
}
if (0 < specialAvailable) {
  i <- 0;
  while (i < policy.`specialMin) {
    specialAvailable <- specialAvailable - 1;
    randomChar <@ random_char_generator(specialSet);
    generatedPassword <- generatedPassword ++ [randomChar];
    i <- i + 1;
  }
}
(* generate characters from the available sets of characters *)
unionSet <@ define_union_set(lowercaseAvailable, uppercaseAvailable, numbersAvailable,
                  specialAvailable, lowercaseSet, uppercaseSet, numbersSet, specialSet);
while (size generatedPassword < policy.`length) {
  randomChar <@ random_char_generator(unionSet);
  if (randomChar \in lowercaseSet) {
    lowercaseAvailable <- lowercaseAvailable - 1;
```

```
        if (lowercaseAvailable = 0) {
          unionSet <@ define_union_set(lowercaseAvailable, uppercaseAvailable, numbersAvailable,
                                       specialAvailable, lowercaseSet, uppercaseSet, numbersSet,
                                       specialSet);
        }
      }
      elif (randomChar \in uppercaseSet) {
        uppercaseAvailable <- uppercaseAvailable - 1;
        if (uppercaseAvailable = 0) {
          unionSet <@ define_union_set(lowercaseAvailable, uppercaseAvailable, numbersAvailable,
                                       specialAvailable, lowercaseSet, uppercaseSet, numbersSet,
                                       specialSet);
        }
      }
      elif (randomChar \in numbersSet) {
        numbersAvailable <- numbersAvailable - 1;
        if (numbersAvailable = 0) {
          unionSet <@ define_union_set(lowercaseAvailable, uppercaseAvailable, numbersAvailable,
                                       specialAvailable, lowercaseSet, uppercaseSet, numbersSet,
                                       specialSet);
        }
      }
      elif (randomChar \in specialSet) {
        specialAvailable <- specialAvailable - 1;
        if (specialAvailable = 0) {
          unionSet <@ define_union_set(lowercaseAvailable, uppercaseAvailable, numbersAvailable,
                                       specialAvailable, lowercaseSet, uppercaseSet, numbersSet,
                                       specialSet);
        }
      }
      generatedPassword <- generatedPassword ++ [randomChar];
    }
      generatedPassword <@ permutation(generatedPassword);
      outputPassword <- Some generatedPassword;
  }
  (* If policy is unsatisfiable, return None *)
  else {
    outputPassword <- None;
  }
  return outputPassword;
}
```

Reference Implementation's Random Char Generator:

```
proc random_char_generator(set:charSet) : char = {
  var char : char;
  var choice : int;

  choice <@ rng(size set);
  char <- nth (-1) set choice;
  return (char);
}
```

Reference Implementation's String Permutation:

```
proc permutation(pw:password) : password = {
  var i : int;
  var j : int;
  var aux : char;

  i <- size pw;
  while (0 < i) {
    j <@ rng(i);
    i <- i - 1;
    aux <- nth 0 pw i;
    pw <- update (nth 0 pw j) pw i;
    pw <- update aux pw j;
  }
  return pw;
}
```

Reference Implementation's Random Number Generator:

```
proc rng(range:int) : int = {
  var value, maxValue, modValue : int;
  (* check how much we should remove from 2^64 - 1 in order to get a multiple of range *)
  modValue <- (2^64 - 1) %% range;
  (* if the mod is range - 1, it means we can sample from 2^64 - 1 and have an uniform distribution *)
  if(modValue = range - 1) {
    maxValue <- (2^64 - 1);
  (* else, we need to remove the unnecessary values*)
  } else {
    maxValue <- (2^64 - 1) - modValue - 1;
  }
  value <$ [0 .. (2^64 - 1)];
  while (maxValue < value) {
    value <$ [0 .. (2^64 - 1)]; (* Random Bytes Generator *)
  }
```

```
    value <- value %% range;
    return value;
  }
```

Reference Implementation's Define Union Set:

```
  proc define_union_set(nLowercase:int, nUppercase:int, nNumbers:int, nSpecial:int,
                        lowercaseSet:charSet, uppercaseSet:charSet,
                        numbersSet:charSet, specialSet:charSet) : charSet = {
    var unionSet, set : charSet;
    unionSet <- [];
    if (0 < nLowercase) {
      set <- lowercaseSet;
      unionSet <- unionSet ++ set;
    }
    if (0 < nUppercase) {
      set <- uppercaseSet;
      unionSet <- unionSet ++ set;
    }
    if (0 < nNumbers) {
      set <- numbersSet;
      unionSet <- unionSet ++ set;
    }
    if (0 < nSpecial) {
      set <- specialSet;
      unionSet <- unionSet ++ set;
    }
    return unionSet;
  }
```

Concrete Scheme (Program used to prove equivalence between the reference implementation and the Jasmin implementation):

```
module ConcreteScheme : RPG_T = {
  proc policySpecToMem(policy:policy, mem:global_mem_t, addr:W64.t) : global_mem_t = {
    mem <- storeW64 mem (W64.to_uint addr + 0) (W64.of_int policy.`length);
    mem <- storeW64 mem (W64.to_uint addr + 8) (W64.of_int policy.`lowercaseMin);
    mem <- storeW64 mem (W64.to_uint addr + 16) (W64.of_int policy.`lowercaseMax);
    mem <- storeW64 mem (W64.to_uint addr + 24) (W64.of_int policy.`uppercaseMin);
    mem <- storeW64 mem (W64.to_uint addr + 32) (W64.of_int policy.`uppercaseMax);
    mem <- storeW64 mem (W64.to_uint addr + 40) (W64.of_int policy.`numbersMin);
    mem <- storeW64 mem (W64.to_uint addr + 48) (W64.of_int policy.`numbersMax);
    mem <- storeW64 mem (W64.to_uint addr + 56) (W64.of_int policy.`specialMin);
    mem <- storeW64 mem (W64.to_uint addr + 64) (W64.of_int policy.`specialMax);
    return mem;
```

```
  }

  proc pwdMemToSpec(length:int, mem:global_mem_t, addr:W64.t) : password = {
    var pwd;
    var i;
    pwd <- [];
    i <- 0;
    while(i < length) {
      pwd <- pwd ++ [W8.to_uint (loadW8 mem (W64.to_uint addr + i))];
    }
    return pwd;
  }

  proc generate_password(policy:policy) : password option = {
    var policyAddr, pwdAddr : W64.t;
    var output : W64.t;
    var pwd : password;
    var pwdOpt : password option;
    (* arbitrary memory location for policy and output password *)
    policyAddr <- W64.of_int 0;
    pwdAddr <- W64.of_int 1000;
    policySpecToMem(policy, Glob.mem, policyAddr);
    output <- M.generate_password(policyAddr, pwdAddr);
    if (output \slt W64.zero) {
      pwdOpt <- None;
    } else {
      pwd <- pwdMemToSpec(policy.`length, Glob.mem, pwdAddr);
      pwdOpt <- Some pwd;
    }
    return pwdOpt;
  }
}.
```

B

Jasmin

Jasmin Implementation:

```
inline fn rng(reg u64 range) -> reg u64 {
    reg u64 max_value, tmp1, tmp2, tmp_range;
    tmp1 = 18446744073709551615; // 2^64 -1
    tmp2 = 18446744073709551615;
    tmp_range = range;
    tmp1 = tmp1 % tmp_range;
    tmp_range -= 1;
    if(tmp1 == tmp_range) {
        max_value = tmp2;
    } else {
        max_value = tmp2 - tmp1;
        max_value -= 1;
    }
    tmp2 = #RDRAND();
    while(max_value < tmp2) {
        tmp2 = #RDRAND();
    }
```

```
    tmp1 = tmp2;
    tmp_range += 1;
    tmp1 = tmp1 % tmp_range;
    return tmp1;

}

inline fn random_char_generator(reg u64 range, stack u8[76] set) -> reg u8 {
    reg u8 char;
    reg u64 choice;
    choice = rng(range);
    char = set[(int)choice];
    return char;
}

inline fn permutation(reg u64 p_string, reg u64 string_len) {
    reg u64 i, j;
    reg u8 aux;
    i = string_len;
    while (i > 0) {
        j = rng(i);
        i = i - 1;
        aux = (u8) [p_string + i];
        (u8) [p_string + i] = (u8) [p_string + j];
        (u8) [p_string + j] = aux;
    }
}

inline fn define_union_set(reg u64 lowercase_max, reg u64 uppercase_max, reg u64 numbers_max, reg u64 special_max, stack u8[
    stack u8[76] uppercase_set, stack u8[76] numbers_set, stack u8[76] special_set, stack u8[76] union_set) -> reg u64, stac

    reg u8 tmp;
    reg u64 i, i_set;
    i_set = 0;
    if (lowercase_max > 0) {
        // i = 0;
        // while (i < 26) {
        //     tmp = lowercase_set[(int)i];
        //     union_set[(int)i_set] = tmp;
        //     i += 1;
        //     i_set += 1;
        // }
        tmp = lowercase_set[0];
        union_set[i_set + 0] = tmp;
        tmp = lowercase_set[1];
```

```
union_set[i_set + 1] = tmp;
tmp = lowercase_set[2];
union_set[i_set + 2] = tmp;
tmp = lowercase_set[3];
union_set[i_set + 3] = tmp;
tmp = lowercase_set[4];
union_set[i_set + 4] = tmp;
tmp = lowercase_set[5];
union_set[i_set + 5] = tmp;
tmp = lowercase_set[6];
union_set[i_set + 6] = tmp;
tmp = lowercase_set[7];
union_set[i_set + 7] = tmp;
tmp = lowercase_set[8];
union_set[i_set + 8] = tmp;
tmp = lowercase_set[9];
union_set[i_set + 9] = tmp;
tmp = lowercase_set[10];
union_set[i_set + 10] = tmp;
tmp = lowercase_set[11];
union_set[i_set + 11] = tmp;
tmp = lowercase_set[12];
union_set[i_set + 12] = tmp;
tmp = lowercase_set[13];
union_set[i_set + 13] = tmp;
tmp = lowercase_set[14];
union_set[i_set + 14] = tmp;
tmp = lowercase_set[15];
union_set[i_set + 15] = tmp;
tmp = lowercase_set[16];
union_set[i_set + 16] = tmp;
tmp = lowercase_set[17];
union_set[i_set + 17] = tmp;
tmp = lowercase_set[18];
union_set[i_set + 18] = tmp;
tmp = lowercase_set[19];
union_set[i_set + 19] = tmp;
tmp = lowercase_set[20];
union_set[i_set + 20] = tmp;
tmp = lowercase_set[21];
union_set[i_set + 21] = tmp;
tmp = lowercase_set[22];
union_set[i_set + 22] = tmp;
tmp = lowercase_set[23];
union_set[i_set + 23] = tmp;
```

```
        tmp = lowercase_set[24];
        union_set[i_set + 24] = tmp;
        tmp = lowercase_set[25];
        union_set[i_set + 25] = tmp;
        i_set += 26;
    }
    if (uppercase_max > 0) {
        i = 0;
        while (i < 26) {
            tmp = uppercase_set[(int)i];
            union_set[(int)i_set] = tmp;
            i += 1;
            i_set += 1;
        }
    }
    if (numbers_max > 0) {
        i = 0;
        while (i < 10) {
            tmp = numbers_set[(int)i];
            union_set[(int)i_set] = tmp;
            i += 1;
            i_set += 1;
        }
    }
    if (special_max > 0) {
        i = 0;
        while (i < 14) {
            tmp = special_set[(int)i];
            union_set[(int)i_set] = tmp;
            i += 1;
            i_set += 1;
        }
    }
    return i_set, union_set;
}

export fn generate_password(reg u64 policy_addr, reg u64 output_addr) -> reg u64 {
    // General purpose registers
    reg u8 tmp8;
    reg u64 tmp64_1, tmp64_2;
    // Registers used for iterations
    reg u64 i, i_set, i_filled;
    // Stack arrays that store the different char sets
    stack u8[76] lowercase_set;
    stack u8[76] uppercase_set;
```

```
stack u8[76] numbers_set;
stack u8[76] special_set;
stack u8[76] union_set;
// Variables that stor min and max values of characters to be generated
stack u64 length, lowercase_min, uppercase_min, numbers_min, special_min, lowercase_max, uppercase_max, numbers_max, spe
// Output
reg u64 output;
////////////////////////////////      Sets of characters     ////////////////////////////////////
//                                                                                              //
//  Lowercase: {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}   //
//  Uppercase: {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z}   //
//  Numbers: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}                                                     //
//  Special Characters: {!, ?, #, $, %, &, +, -, *, _, @, :, ;, =}                              //
//                                                                                              //
//////////////////////////////////////////////////////////////////////////////////////////////////
// Get values from the policy
length          = [policy_addr + 0];
lowercase_min   = [policy_addr + 8];
lowercase_max   = [policy_addr + 16];
uppercase_min   = [policy_addr + 24];
uppercase_max   = [policy_addr + 32];
numbers_min     = [policy_addr + 40];
numbers_max     = [policy_addr + 48];
special_min     = [policy_addr + 56];
special_max     = [policy_addr + 64];

//////////////////////////
// INPUT VERIFICATIONS //
//////////////////////////
// Length does not exceed maximum length of 200
tmp64_1 = length;
if (tmp64_1 <= 200) {
// Length is greater than zero
if (tmp64_1 > 0) {
// Min values are not negative
tmp64_1 = lowercase_min;
if (tmp64_1 >= 0) {
tmp64_1 = uppercase_min;
if (tmp64_1 >= 0) {
tmp64_1 = numbers_min;
if (tmp64_1 >= 0) {
tmp64_1 = special_min;
if (tmp64_1 >= 0) {
// Max values do not exceed 200 (theoretically this is useless, but we need this in the implementation to avoid overflow
tmp64_1 = lowercase_max;
```

```
if (tmp64_1 <= 200) {
tmp64_1 = uppercase_max;
if (tmp64_1 <= 200) {
tmp64_1 = numbers_max;
if (tmp64_1 <= 200) {
tmp64_1 = special_max;
if (tmp64_1 <= 200) {
// Max values are greater or equal to min values
tmp64_1 = lowercase_max;
tmp64_2 = lowercase_min;
if (tmp64_1 >= tmp64_2) {
tmp64_1 = uppercase_max;
tmp64_2 = uppercase_min;
if (tmp64_1 >= tmp64_2) {
tmp64_1 = numbers_max;
tmp64_2 = numbers_min;
if (tmp64_1 >= tmp64_2) {
tmp64_1 = special_max;
tmp64_2 = special_min;
if (tmp64_1 >= tmp64_2) {
// Sum of minimum values do not exceed password length
tmp64_1 = lowercase_min;
tmp64_2 = uppercase_min;
tmp64_1 += tmp64_2;
tmp64_2 = numbers_min;
tmp64_1 += tmp64_2;
tmp64_2 = special_min;
tmp64_1 += tmp64_2;
if (tmp64_1 <= length) {
// Sum of maximum values satisfies the password length
tmp64_1 = lowercase_max;
tmp64_2 = uppercase_max;
tmp64_1 += tmp64_2;
tmp64_2 = numbers_max;
tmp64_1 += tmp64_2;
tmp64_2 = special_max;
tmp64_1 += tmp64_2;
if (tmp64_1 >= length) {

////////////////////////////////
// READY TO GENERATE PASSWORD //
////////////////////////////////
// Initialize the sets. The values are the ASCII codes of the different characters
lowercase_set[0] = 97;
lowercase_set[1] = 98;
```

```
lowercase_set[2] = 99;
lowercase_set[3] = 100;
lowercase_set[4] = 101;
lowercase_set[5] = 102;
lowercase_set[6] = 103;
lowercase_set[7] = 104;
lowercase_set[8] = 105;
lowercase_set[9] = 106;
lowercase_set[10] = 107;
lowercase_set[11] = 108;
lowercase_set[12] = 109;
lowercase_set[13] = 110;
lowercase_set[14] = 111;
lowercase_set[15] = 112;
lowercase_set[16] = 113;
lowercase_set[17] = 114;
lowercase_set[18] = 115;
lowercase_set[19] = 116;
lowercase_set[20] = 117;
lowercase_set[21] = 118;
lowercase_set[22] = 119;
lowercase_set[23] = 120;
lowercase_set[24] = 121;
lowercase_set[25] = 122;
i = 26;
while (i < (76)) {
    lowercase_set[(int)i] = 0;
    i += 1;
}
uppercase_set[0] = 65;
uppercase_set[1] = 66;
uppercase_set[2] = 67;
uppercase_set[3] = 68;
uppercase_set[4] = 69;
uppercase_set[5] = 70;
uppercase_set[6] = 71;
uppercase_set[7] = 72;
uppercase_set[8] = 73;
uppercase_set[9] = 74;
uppercase_set[10] = 75;
uppercase_set[11] = 76;
uppercase_set[12] = 77;
uppercase_set[13] = 78;
uppercase_set[14] = 79;
uppercase_set[15] = 80;
```

```
uppercase_set[16] = 81;
uppercase_set[17] = 82;
uppercase_set[18] = 83;
uppercase_set[19] = 84;
uppercase_set[20] = 85;
uppercase_set[21] = 86;
uppercase_set[22] = 87;
uppercase_set[23] = 88;
uppercase_set[24] = 89;
uppercase_set[25] = 90;
i = 26;
while (i < (76)) {
    uppercase_set[(int)i] = 0;
    i += 1;
}
numbers_set[0] = 48;
numbers_set[1] = 49;
numbers_set[2] = 50;
numbers_set[3] = 51;
numbers_set[4] = 52;
numbers_set[5] = 53;
numbers_set[6] = 54;
numbers_set[7] = 55;
numbers_set[8] = 56;
numbers_set[9] = 57;
i = 10;
while (i < (76)) { // Maybe remove this whiles
    numbers_set[(int)i] = 0;
    i += 1;
}
special_set[0] = 33;
special_set[1] = 63;
special_set[2] = 35;
special_set[3] = 36;
special_set[4] = 37;
special_set[5] = 38;
special_set[6] = 43;
special_set[7] = 45;
special_set[8] = 42;
special_set[9] = 95;
special_set[10] = 64;
special_set[11] = 58;
special_set[12] = 59;
special_set[13] = 61;
i = 14;
```

```
while (i < (76)) {
    special_set[(int)i] = 0;
    i += 1;
}
// Initialize union set
i = 0;
while (i < (76)) {
    union_set[(int)i] = 0;
    i += 1;
}
// Generate characters with min values defined
i_filled = 0;
if (lowercase_max > 0) {
    i = 0;
    while (i < lowercase_min) {
        lowercase_max -= 1;
        tmp8 = random_char_generator(26, lowercase_set);
        (u8) [output_addr + i_filled] = tmp8;
        i += 1;
        i_filled += 1;
    }
}
if (uppercase_max > 0) {
    i = 0;
    while (i < uppercase_min) {
        uppercase_max -= 1;
        tmp8 = random_char_generator(26, uppercase_set);
        (u8) [output_addr + i_filled] = tmp8;
        i += 1;
        i_filled += 1;
    }
}
if (numbers_max > 0) {
    i = 0;
    while (i < numbers_min) {
        numbers_max -= 1;
        tmp8 = random_char_generator(10, numbers_set);
        (u8) [output_addr + i_filled] = tmp8;
        i += 1;
        i_filled += 1;
    }
}
if (special_max > 0) {
    i = 0;
    while (i < special_min) {
```

```
        special_max -= 1;
        tmp8 = random_char_generator(14, special_set);
        (u8) [output_addr + i_filled] = tmp8;
        i += 1;
        i_filled += 1;
    }
}
// Define set of available characters, if their maximum value has not been reached yet
tmp64_1, union_set = define_union_set(lowercase_max, uppercase_max, numbers_max, special_max,
                    lowercase_set, uppercase_set, numbers_set, special_set, union_set);

// Generate password from the set of available characters
tmp64_2 = length;
while (i_filled < tmp64_2) {
    tmp8 = random_char_generator(tmp64_1, union_set);
    if (tmp8 >= 97) { if (tmp8 <= 122) {
            lowercase_max -= 1;
            if (lowercase_max == 0) {
                tmp64_1, union_set = define_union_set(lowercase_max, uppercase_max, numbers_max, special_max,
                                    lowercase_set, uppercase_set, numbers_set, special_set, union_set);
            }
    }} else { if (tmp8 >= 65) { if (tmp8 <= 90) {
            uppercase_max -= 1;
            if (uppercase_max == 0) {
                tmp64_1, union_set = define_union_set(lowercase_max, uppercase_max, numbers_max, special_max,
                                    lowercase_set, uppercase_set, numbers_set, special_set, union_set);
            }
        }} else { if (tmp8 >= 48) { if (tmp8 <= 57) {
                numbers_max -= 1;
                if (numbers_max == 0) {
                    tmp64_1, union_set = define_union_set(lowercase_max, uppercase_max, numbers_max, special_max,
                                        lowercase_set, uppercase_set, numbers_set, special_set, union_set);
                }
            }} else {
                special_max -= 1;
                if (special_max == 0) {
                    tmp64_1, union_set = define_union_set(lowercase_max, uppercase_max, numbers_max, special_max,
                                        lowercase_set, uppercase_set, numbers_set, special_set, union_set);
                }
            }
        }
    }
    (u8) [output_addr + i_filled] = tmp8;
    i_filled += 1;
}
```

```
// Generate random permutation of string
tmp64_1 = length;
permutation(output_addr, tmp64_1);
// Make sure last character is \0
[output_addr + tmp64_1] = 0;
output = 1;
} else {
    // Maximum values are short (Sum(Max) < length)
    output = -16;
}
} else {
    // Minimum values are larger than length (Sum(Min) > length)
    output = -15;
}
} else {
    // Special characters max value is smaller than min value
    output = -14;
}
} else {
    // Numbers max value is smaller than min value
    output = -13;
}
} else {
    // Uppercase letters max value is smaller than min value
    output = -12;
}
} else {
    // Lowercase letters max value is smaller than min value
    output = -11;
}
} else {
    //  Special characters max value is greater than 200
    output = -10;
}
} else {
    //  Numbers max value is greater than 200
    output = -9;
}
} else {
    // Uppercase letters max value is greater than 200
    output = -8;
}
} else {
    // Lowercase letters max value is greater than 200
    output = -7;
```

```
    }
    } else {
        // Special characters min value is negative
        output = -6;
    }
    } else {
        // Numbers min value is negative
        output = -5;
    }
    } else {
        // Uppercase letters min value is negative
        output = -4;
    }
    } else {
        // Lowercase letters min value is negative
        output = -3;
    }
    } else {
        // Length is negative
        output = -2;
    }
    } else {
        // Length is too large (length > 200)
        output = -1;
    }
    return output;
}
```