Towards Formal Verification of Password Generation Algorithms used in Password Managers*

Miguel Grilo[1], João F. Ferreira[2], and José Bacelar Almeida[3]

[1] INESC TEC and IST, University of Lisbon, Portugal
[2] INESC-ID and IST, University of Lisbon, Portugal
[3] HASLab, INESC TEC and University of Minho, Portugal

Abstract. Password managers are important tools that enable us to use stronger passwords, freeing us from the cognitive burden of remembering them. Despite this, there are still many users who do not fully trust password managers. In this paper, we focus on a feature that most password managers offer that might impact the user's trust, which is the process of generating a random password. We survey which algorithms are most commonly used and we propose a solution for a formally verified reference implementation of a password generation algorithm. We use EasyCrypt as our framework to both specify the reference implementation and to prove its functional correctness and security.

Keywords: Password Manager · Random Password Generator · Formal Verification · Security

1 Introduction

To address many of the existing problems and vulnerabilities regarding password authentication [6, 7, 9, 10, 12, 15], security experts recommend using password managers (PMs) for both storing and generating strong random passwords [8]. However, despite these recommendations, users tend to reject PMs, partly because they do not fully trust these applications [1, 3, 11]. It has been observed that generation of random passwords is one important feature that increases use of PMs [1] and helps prevent the use of weaker passwords and password reuse [11]. These studies suggest that a strong password generator that users can fully trust is a must-have feature for PMs.

In this paper, we propose a formally verified reference implementation for a Random Password Generator (RPG). We also present the formalization of functional correctness and security properties of our implementation, using the EasyCrypt proof environment [4] and the game-based approach for cryptographic security proofs [5, 14].

* Supported by the PassCert project, a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019. Project URL: `https://passcert-project.github.io`

2 Current Password Generation Algorithms

We studied 15 PMs to understand what are the most commonly used algorithms, but in this paper we focus on three of them: Google Chrome's PM (v89.0.4364.1)[4], Bitwarden (v1.47.1)[5], and KeePass (v2.46)[6]. These were chosen because they are widely used and open-source, which allows us to access their source code and study them in detail.

2.1 Password Composition Policies

In general, PMs allow users to define password composition policies that the generated passwords must satisfy. These policies define the structure of the password, including the password's length and the different character classes that may be used. These policies are used to restrict the space of user-created passwords, thus precluding some easily guessed passwords. Table 1 shows the policies that can be specified in the studied PMs.

	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
Define minimum and maximum occurrence 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 1. Available policy options a user can define. The Alphabetic set in Chrome is the union of Lowercase Letters and Uppercase Letters. The set of Special Characters in Chrome and Bitwarden is {- _ . : !} while in KeePass it is {! " # $ % & ' * + , . / : ; = ? @ \ ^ |}. The Brackets set in KeePass is {( ) { } [ ] ⟨⟩}. The Space, Minus, and Underline are the single element sets {␣}, {-}, and {_}, respectively.

[4] https://source.chromium.org/chromium/chromium/src/+/master:components
[5] https://github.com/bitwarden
[6] https://github.com/dlech/KeePass2.x

2.2 Random Password Generation

The main idea of the surveyed algorithms is to generate random characters from the different character sets until the password length is fulfilled, taking also into consideration the minimum and maximum occurrences of characters per set. Chrome's algorithm starts by randomly generating characters from the sets which have the minimum number of occurrences defined. Then, it generates characters from the union of all sets which have not already reached their maximum number of occurrences. Lastly, it generates a permutation on the characters of the string, resulting in a random generated password. Bitwarden's algorithm is similar, but it makes the permutation before generating the characters (i.e., it creates a string like '*llunl*' to express that the first two characters are lowercase letters, followed by an uppercase letter, then a number, and finally a lowercase letter. Only then it generates the characters from the respective sets). KeePass does not support defining the minimum and maximum occurrences of characters per set, so the algorithm just randomly generates characters from the union of the sets defined in the policy.

2.3 String Permutation

Given the need to generate a random permutation of the characters of a string, Bitwarden and Chrome both implement an algorithm to do so. The basic idea for both PMs is the same, which is to randomly choose one character from the original string for each position of the new string.

2.4 Random Number Generator

The RPG needs to have an implementation of a Random Number Generator (RNG) that generates random numbers within a range of values. Chrome and KeePass use similar RNGs that generate numbers from 0 to an input *range*. The main idea of these two PMs is to generate random bytes, then casting them to an integer, and then return that value modulo range, so the value it generates is between 0 and range. Bitwarden's RNG allows generating numbers from an arbitrary minimum value up to an arbitrary maximum value. Since in our algorithm we do not need this feature, the simpler implementations of Chrome and KeePass are sufficient for us to focus on.

All these RNGs call a random bytes generator. Regarding this, the three PMs considered use different approaches. Chrome uses system calls depending on the operating system it is running, Bitwarden uses the NodeJS *randomBytes()* method, while KeePass defines its own random bytes generator based on ChaCha20. In this work, we assume that random bytes generator exists and are secure.

3 Reference Implementation

Based on our survey (Section 2), we propose a reference implementation for an RPG which offers the following policy adjustments: (1) the user can define

the password length (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. The restriction on the maximum length is just to avoid passwords that are unnecessarily large, since passwords with at least 16 characters are already hard to guess and secure [13]. In Algorithm 1 we show the pseudo-code of the proposed reference implementation.

Our algorithm receives as input the password composition policy. Then it 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, until the size of *password* is equal to the length defined in the policy, it randomly generates characters from the union of all sets which have not more than their *max* value of characters in *password*. Finally, it generates a random permutation of the string, and returns it.

4 Formal Proofs

In this section we present our two main properties to be proved about our RPG: functional correctness and security.

4.1 Functional Correctness

We say that an RPG is functionally correct if generated passwords satisfy the input policy (the probability that the generated password will satisfy the policy is 1). This property guarantees that users will always get an output according to their expectations. In the following code we show a succinct version of the formalization of functional correctness in EasyCrypt style.

```
module Correctness(RPG : RPG_T) = {

    proc main(policy:policy) : bool = {
        var password : password;
        var satLength, satBounds : bool;
        password <@ RPG.generate_password(policy);
        satLength <@ satisfies_length(password, policy);
        satBounds <@ satisfies_bounds(password, policy);
        return satLength /\ satBounds;
    }
}.
```

The procedures `satisfies_length` and `satisfies_bounds` check, respectively, if the password's length is the same as the one defined in the policy and if the *max* and *min* values defined per set in the policy are satisfied.

Using this definition, we can define the lemma `correctness_phl` which can be proved using probabilistic Hoare logic (pHL):

```
lemma correctness_phl (p:policy) :
  Pr[Correctness(RPGRef).main : policy = p ==> res] = 1%r.
```

Algorithm 1 RPG Reference Implementation

```
1: procedure GENERATE(policiy)
2:     pwLength ← policy.pwLength
3:     charSets ← policy.charSets
4:     password ← ε
5:     for all set ∈ charSets do
6:         for i = 1, 2, ..., set.min do
7:             char ← GENERATECHARACTER(set)
8:             password ← password||char
9:         end for
10:    end for
11:    while len(password) < pwLength do
12:        availableSets ← ⋃_{set∈charSets} set such that set.max > 0
13:        char ← GENERATECHARACTER(availableSets)
14:        password ← password||char
15:    end while
16:    password ← PERMUTATION(password)
17:    return password
18: end procedure
19:
20: procedure GENERATECHARACTER(set)
21:    choice ← RNG(set.size)
22:    set.max ← set.max − 1
23:    return choice
24: end procedure
25:
26: procedure PERMUTATION(string)
27:    for i = len(string) − 1, ..., 0 do
28:        j ← RNG(i)
29:        aux = string[i]
30:        string[i] = string[j]
31:        string[j] = aux
32:    end for
33:    return string
34: end procedure
35:
36: procedure RNG(range)
37:    maxValue ← (uint64.maxValue/range) ∗ range − 1
38:    do
39:        value ← (uint64) GenerateRandomBytes
40:    while value > maxValue
41:    return value mod range
42: end procedure
```

4.2 Security

Regarding security, we want to verify that, given the policy, the generated password has the same probability of being generated as any other possible password. In other words, considering the entire set of possible passwords defined by the policies, we want to make sure that there is a uniform distribution over that set. To prove such property we can use the game-based approach for cryptographic security proofs [5, 14]. This methodology is supported by EasyCrypt.

The notion of security for an RPG is expressed by two games $\text{Real}_{\mathcal{A}}()$ and $\text{Ideal}_{\mathcal{A}}()$, which are parameterized by an attacker $\mathcal{A}$. This attacker has access to an oracle in each game (one for our reference implementation and one for the ideal RPG). The attacker is trying to guess with which oracle he is interacting with, by outputting a boolean b. The games are shown in Figure 1.

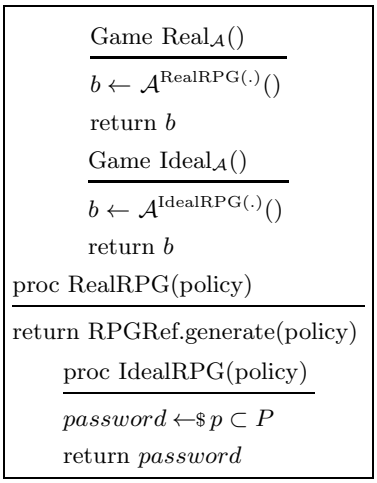

Fig. 1. Security games. RPGRef is our reference implementation and p is the subset of the set of all possible passwords P that satisfy the given policy.

In order to consider our implementation secure, the attacker can not be able do distinguish between it and the ideal RPG. This means that the following advantage must be neglibible

$$\mathsf{Adv}^{\text{rpg}}(\mathcal{A}) = \Pr[\text{Real}_{\mathcal{A}}() \Rightarrow \text{res}] \text{ - } \Pr[\text{Ideal}_{\mathcal{A}}() \Rightarrow \text{res}]$$

where $\Pr[game() \Rightarrow \text{res}]$ is the probability that $game()$ outputs true.

5 Conclusion

In this paper we presented an analysis of the RPG algorithms currently used in popular PMs, and we proposed a reference implementation. Then, we showed the formalization of two properties that we can verify using EasyCrypt: its functional correctness and security.

As future work, after proving the properties, we will implement the reference using Jasmin [2], a framework for developing high-speed and high-assurance cryptographic software. We might also extend the policy composition options with, for example, the option for the user to define a character set "by hand". While generally speaking strict password composition policies are a good security mechanism, these can still generate some easily guessed passwords (e.g., a policy that enforces the use of all character classes may generate the easily guessed password "P@ssw0rd") [13]. So, it might also be interesting to define and formalize a property regarding password strength, which would guarantee that our RPG would only generate strong passwords (according to some metric).

References

[1] Nora Alkaldi and Karen Renaud. "Why do people adopt, or reject, smartphone password managers?" In: (2016).

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves 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.

[3] Salvatore Aurigemma, Thomas Mattson, and Lori Leonard. "So much promise, so little use: What is stopping home end-users from using password manager applications?" In: (2017).

[4] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. "Easycrypt: A tutorial". In: *Foundations of security analysis and design vii*. Springer, 2013, pp. 146–166.

[5] Mihir Bellare and Phillip Rogaway. "Code-Based Game-Playing Proofs and the Security of Triple Encryption." In: *IACR Cryptol. ePrint Arch.* 2004 (2004), p. 331.

[6] João F. Ferreira, Saul Johnson, Alexandra Mendes, and Phillip 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.

[7] Dinei Florencio and Cormac Herley. "A large-scale study of web password habits". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 657–666.

[8] Troy Hunt. *Passwords evolved: Authentication guidance for the modern era. troyhunt. com*. 2017.

[9] Saul Johnson, João F. Ferreira, Alexandra Mendes, and Julien 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.

[10] Poul-Henning Kamp, P Godefroid, M Levin, D Molnar, P McKenzie, R Stapleton-Gray, B Woodcock, and G Neville-Neil. "LinkedIn password leak: salt their hide." In: *ACM Queue* 10.6 (2012), p. 20.

[11] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith 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] David Pereira, João F. Ferreira, and Alexandra 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.

[13] Richard Shay, Saranga Komanduri, Adam L Durity, Phillip Huh, Michelle L Mazurek, Sean M Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. "Designing password policies for strength and usabil-

ity". In: *ACM Transactions on Information and System Security (TISSEC)* 18.4 (2016), pp. 1–34.

[14] Victor Shoup. "Sequences of games: a tool for taming complexity in security proofs." In: *IACR Cryptol. ePrint Arch.* 2004 (2004), p. 332.

[15] Chaoshun Zuo, Zhiqiang Lin, and Yinqian 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.

A Appendix: Algorithms Discussed in Section 2

A.1 RPG Algorithm

Following the discussion in Section 2, we present the following pseudo-code (Algorithm 2) which is a generalization of the algorithms of the three password managers, being closely similar to Chrome's algorithm.

Algorithm 2 General Password Generation Algorithm

```
1: procedure GeneralGenerate(policies)
2:     pwLength ← policies.pwLength
3:     charSets ← policies.charSets
4:     password ← ε
5:     for all set ∈ charSets do
6:         for i = 1, 2, ..., set.min do          ▷ set.min is the minimum number of characters from that set that must appear on the generated password
7:             char ← Randomly generate a char from the set
8:             password ← password||char          ▷ '||' is the append operator
9:         end for
10:    end for
11:    while len(password) < pwLength do
12:        char ← Randomly generate a char from the union of all sets, while not exceeding the max char occurrence of each set
13:        password ← password||char
14:    end while
15:    Generate a random permutation of password
16:    Output password
17: end procedure
```

A.2 String Permutation

The method described in Section 2.3 is described in pseudo-code in Algorithm 3.

Algorithm 3 String Permutation

```
1: procedure GeneralPermutation(string)
2:     for i = len(string) − 1, ..., 0 do
3:         j ← Random number between 0 and i
4:         aux = string[i]
5:         string[i] = string[j]
6:         string[j] = aux
7:     end for
8: end procedure
```

A.3 Random Number Generator

Chrome and KeePass RNG algorithms described in Section 2.4 are described in pseudo-code in Algorithm 4.

Algorithm 4 RNGs with maximum range

```
1: procedure ChromeRNG(range)
2:     maxValue ← (uint64.maxValue/range) ∗ range − 1      ▷ uint64.maxValue is the maximum value an unsigned integer with 64 bits can take; '/' is the Euclidean division
3:     do
4:         value ← (uint64) GenerateRandomBytes
5:     while value > maxValue
6:     return value mod range
7: end procedure
8:
9: procedure KeePassRNG(range)
10:    do
11:        genValue ← (uint64) GenerateRandomBytes
12:        value ← genValue mod range
13:    while (genValue − value) > (uint64.maxValue − (range − 1))
14:    return value
15: end procedure
```