Verified Password Generation from Password Composition Policies

Miguel Grilo[1], João Campos[2], João F. Ferreira[2], José Bacelar Almeida[3], and Alexandra Mendes[4]

[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
[4] HASLab, INESC TEC and Faculty of Engineering, University of Porto, Portugal

Abstract. Password managers (PMs) are important tools that enable the use of stronger passwords, freeing users from the cognitive burden of remembering them. Despite this, there are still many users who do not fully trust PMs. In this paper, we focus on a feature that most PMs offer that might impact the user's trust, which is the process of generating a random password. We present three of the most commonly used algorithms and we propose a solution for a formally verified reference implementation of a password generation algorithm. We use EasyCrypt to specify and verify our reference implementation. In addition, we present a proof-of-concept prototype that extends Bitwarden to only generate compliant passwords, solving a frequent users' frustration with PMs. This demonstrates that our formally verified component can be integrated into an existing (and widely used) PM.

Keywords: Password Manager · Random Password Generator · Formal Verification · Security · EasyCrypt · Jasmin · Interactive Theorem Proving · Verified Compilation · Bitwarden

1 Introduction

To address many of the existing problems regarding password authentication [16, 22, 28], security experts often recommend using password managers (PMs) for storing and generating strong random passwords. Indeed, a key feature of PMs is random password generation, since it helps prevent the use of weaker passwords and password reuse [21]. Moreover, it provides users with a greater sense of security [1], thus contributing to a wider adoption of PMs.

However, users frequently express concern and disapproval when PMs do not generate passwords compliant [6, 26] with the password composition policies stipulated by the services they use [12]. Stajano et al. [25] argue that this problem arises due to very restrictive password composition policies that services usually have [13]. These policies present a greater challenge to password managers since randomly generated passwords have a higher chance of being non-compliant with more restrictive policies.

This problem leads to frustrated users and can be an obstacle to the adoption of PMs. Therefore, it is important to ensure that the password generation component of a PM is reliable. In particular, it is desirable to guarantee that generated passwords (1) satisfy the requirements specified by the user (or service), and (2) are uniformly sampled from the universe induced by the password policy, thus guaranteeing unpredictability of the password generator. In this paper, we propose a formally verified reference implementation for a Random Password Generator (RPG) that addresses these two points. Our main contributions are:

1. We use EasyCrypt [7] to prove that all the passwords generated by our reference implementation satisfy the given password composition policy and that when the given policy is unsatisfiable, the implementation does not generate any password.
2. We formalize the security property stating that our reference implementation samples the set of passwords according to a uniform distribution, using the game-based approach for cryptographic security proofs [8, 24]. This justifies the use of EasyCrypt, since we the need to reason about probability distributions.
3. We extend the open-source PM Bitwarden to (1) read Apple's Password Autofill Rules [5] and to (2) generate passwords using a Jasmin [2] implementation provably equivalent to our reference implementation. This case study is a proof-of-concept that integrates interactive theorem proving (EasyCrypt) and verified compilation (Jasmin) to solve an existing frustration with PMs generating non-compliant passwords. It also demonstrates that our formally verified component can be integrated into an existing (and widely used) PM[5]. Part of this extension was submitted to the Bitwarden team, who accepted it and will merge it into the product after a process of code review.

After reviewing current password generation algorithms in Section 2, we present our reference implementation and its verification in Section 3. In Section 4 we present the end-to-end case study and in Section 5 we discuss related work. We conclude the paper in Section 6, where we also discuss future work.

2 Current Password Generation Algorithms

In this section we present a brief description of widely-used password generation algorithms. We focus on the password generation algorithms of three PMs: Google Chrome's PM (v89.0.4364.1)[6], Bitwarden (v1.47.1)[7], and KeePass (v2.46)[8]. These were chosen because they are widely used and open-source, which allows us to access their source code and study them in detail.

[5] `https://github.com/passcert-project/pw_generator_server`
[6] `https://source.chromium.org/chromium/chromium/src/+/master:components`
[7] `https://github.com/bitwarden`
[8] `https://github.com/dlech/KeePass2.x`

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 its 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 that are easily guessed. Table 1 shows the policies that can be specified in the studied PMs. In the second row, 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.

	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 1. Available policy options a user can define.

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. For example, it starts by creating 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.

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.

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*. Bitwarden's RNG allows generating numbers from an arbitrary minimum value up to an arbitrary maximum value, but it can trivially be reduced to the former approach. The main idea of these RNGs is (1) to rely on a random byte generator, (2) to perform some form of rejection sampling to ensure uniformly distributed values up to a given bound, and (3) finally reducing it to the required range.

The three PMs adopt different approaches regarding the source of random bytes: 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. Because of these different strategies, *in this work we choose not to address the pseudo-random nature of the random byte generator — instead, we assume the existence of a mechanism allowing to sample bytes according to an uniform distribution.* Specifically, we assume an operation that uniformly samples 64-bit words, and then reason on the remaining steps towards the construction of an arbitrary integer range RNG.

3 Verified Password Generation

In this section, we present our reference implementation and the properties that we formally prove. We separate the specifications into an abstract overview, followed by a concrete one in EasyCrypt.

3.1 Reference Implementation

Abstract Overview. Based on the information presented in 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 the same as in Chrome's algorithm (also, we argue that 200 is a reasonable limit, since that arbitrary passwords with at least 16 characters seem to be hard to guess and considered secure [23]).

The pseudo-code of the proposed reference implementation is shown in Algorithm 1. 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. The policy is satisfiable if the defined *length* is in the interval $[1, 200]$, 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.

To output a random generated 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.

EasyCrypt Implementation. EasyCrypt [7] is an interactive framework for verifying the security of cryptographic constructions and protocols using a game-based approach [8, 24]. EasyCrypt implements program logics for proving properties of imperative programs. Its main logics are Hoare Logic and Relational Hoare Logic. Relational Hoare Logic is essential in EasyCrypt, because it provides the ability to establish relations between programs, and how they affect memory values, which is fundamental in the game-based approach. Notice that we do not consider any cryptographic assumption — our use of EasyCrypt is rather justified on the need to reason about probability distributions (e.g. in reasoning on the RNG procedure, as explained above), alongside with more standard Hoare Logic reasoning used for proving correctness assertions.

To model our reference implementation in EasyCrypt, we need to be more precise regarding the types and methods of the algorithm. Figure 1 shows the definitions of the 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 necessary fields to specify a password composition policy. All this information is in a repository in GitHub[9], as well as some other previously explained definitions (e.g., satisfiability of a *policy*), theorems, and properties about these types.

[9] `https://github.com/passcert-project/random-password-generator/blob/main/EC/PasswordGenerationTh.eca`

```
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
}.
```

Fig. 1. Type definitions

Regarding the methods, it is easy to see how the abstract version of the reference implementation maps to the EasyCrypt implementation[10]. The main difference is when defining the *unionSet*. In the abstract implementation, we just say that this variable is the union of all sets such that their *max* values are greater than 0. In EasyCrypt we have the method `define_union_set` which implements this idea. To simplify the proofs, instead of decrementing the *max* value of a set after sampling a character from it, our algorithm has some extra variables (e.g., `lowercaseAvailable` for the Lowercase Set) which say how many characters we can still sample from the respective set. The method `define_union_set` receives these variables as arguments, and defines the union of the sets which we can still sample characters from.

3.2 Formal Proofs

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

Functional Correctness (Abstract). *We say that an RPG is functionally correct if generated passwords satisfy the input policy. This property guarantees that users will always get an output according to their expectations.*

We follow the standard approach of expressing correctness of the scheme by a probabilistic experiment that checks if the specification is fulfilled. Figure 2 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 input policy is satisfiable, it checks if the password satisfies that policy. Otherwise, it returns whether it is equal to *None*. To simplify the reasoning around this property, when the policy is satisfiable, one can separate

Correctness^{RPG}(*policy*)

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

Fig. 2. Correctness Experiment (Abstract)

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

Algorithm 1 RPG Reference Implementation

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

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.

Functional Correctness (EasyCrypt). In EasyCrypt, the correctness experiment is modelled as the module `Correctness`, shown in Figure 3. It is parameterized by a password generator implementation (being `RPG_T` its signature), and has a single method `main` encoding the experiment. 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` – `is_some` and `is_none` are predicates that query the constructor used in an optional value, and `oget` extracts a password from it (if available). The experiment simply executes the RPG and, depending on the satisfiability of the policy, either checks if the generated password satisfies it, or if it is equal to `None`. The EasyCrypt code is available online[11,12].

```
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;
   }
}.
```

Fig. 3. Correctness Procedure (Easycrypt)

The correctness property can be expressed in EasyCrypt as follows:

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

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

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

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 (without any constraint on input policy). The proof of this lemma amounts essentially to prove termination of the `main` method, while also proving 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 `islossless` assertion states that `Correctness(RPGRef).main` terminates with probability 1 for any input. Notice that this is indeed non-trivial, as our RPG performs rejection sampling. Hence, we are not able to prove a concrete bound for the number of iterations for the loop in the `RNG` procedure (Algorithm 1), but we nevertheless establish that it eventually terminates (actually, in expected constant time).

The second lemma is an Hoare triple. In EasyCrypt an Hoare triple is written as `hoare [Command : Precondition ==> Postcondition]`. 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 /\ satisfiesLength p (oget res)
                      /\ satisfiesBounds 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`. As mentioned in Section 3.2, the first lemma can be proved by separately reasoning about the generated password satisfying the length defined in the policy, and then about the different set bounds. 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)
        ].
```

It is easy to see that 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.

Security (Abstract). *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.* In other words, the `generate_password` method samples the set of passwords that satisfy the policy according to a uniform distribution. To prove this property we can use the game-based approach for cryptographic security proofs [8, 24].

As shown abstractly in Figure 4, 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 (here, sampling is denoted by the operator $\leftarrow\$$).

If the policy is not satisfiable, it outputs *None*. 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 *None*
fi

Fig. 4. 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.

Security (EasyCrypt). To construct the IdealRPG module, we start by axiomatizing uniform distributions over the type of *passwords*:

```
op dpassword : password distr.
axiom dpassword_ll : is_lossless dpassword.
axiom dpassword_uni : is_uniform dpassword.
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 (length and admissible chars). Here, `validPassword p` is true if `p` contains only valid characters (lowercase, uppercase, digits, and allowed symbols) and if its length is at most 200. 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 \ (fun pass => !(satisfiesPolicy(policy pass)));
      out <- Some pw;
    } else {
      out <- None;
    }
    return out;
  }
}.
```

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 *dm1* and *dm2*, 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.

General Steps To Prove Security. To prove the security lemma stated above, we need to establish that the induced distribution from the execution of RealRPG is uniform among all passwords satisfying the policy. It requires fairly detailed reasoning on the distribution level in EasyCrypt. The mechanised proof is work in progress; 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 where each character class prescribed in the policy is placed in a specific position (what we have called policy-normalised password); (2) It randomly shuffles the password. The result follows from arguing that policy-normalised passwords are sampled according to a uniform distribution, and that the final shuffle allows to reach 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 (procedure RNG) and the *Fisher-Yates shuffle* algorithm (procedure PERMUTATION) have to be tackled.

4 Case Study: From Apple Password Rules to Verified Password Generation in Bitwarden

This section describes a proof-of-concept prototype that integrates a Jasmin [2] implementation provably equivalent to our reference implementation into a widely-used PM. In particular, we extend Bitwarden to (1) read Apple's Password Autofill Rules [5], which are password composition policies in a format defined by Apple, and (2) to generate compliant passwords using our Jasmin implementation.

The proof-of-concept offers a solution to the common problem of users being concerned and disappointed by the fact that passwords generated by the PM are often not compliant with the password composition policies stipulated by the websites they use [12]. One way to solve this problem is to, first, provide a domain specific language (DSL) that services can use to specify their required password composition policies, and, second, ensure that PMs use the DSL specifications in their password generation algorithms. There have been some proposals for this: Stajano et al. proposed the creation of HTML semantic labels [25] and Horsch et al. proposed the Password Policy Markup Language [18]. Oesch and Ruoti [20] recently reinforced this idea, suggesting that this type of annotations could help the users with using PMs, as well as increase the accuracy of the password

```
<rule> ::= (<required> | <allowed> | <length_reqs> | <max_consecutive>)*
<required> ::= "required: " <list_ids_classes> "; "
<allowed> ::= "allowed: " <list_ids_classes> "; "
<length_reqs> ::= "minlength: " <non_negative_integer> "; "
                | "maxlength: " <non_negative_integer> "; "
<max_consecutive> ::= "max-consecutive: " <non_negative_integer> "; "
<id_class> ::= (<identifier> | <character_class>)
<list_ids_classes> ::= <id_class> | <id_class> ", " <list_ids_classes>
<identifier> ::= "lower" | "upper" | "digits" | "special"
               | "ascii-printable" | "unicode"
<character_class> ::= "[" (<upper> | <lower> | <special> | <digit>)+ "]"
```

Fig. 5. Grammar used by Apple's Password Autofill Rules.

generator. While investigating a way to achieve this with modern PMs, we found that Apple has also developed a DSL to express Password Autofill Rules [5]. The idea is to add a specification to the HTML code, in the form of annotations.

4.1 Apple's Password Autofill Rules

Apple's DSL is based on five properties — *required, allowed, max-consecutive, minlength, and maxlength* — and some identifiers that describe character classes — *upper, lower, digits, special, ascii-printable, and unicode*. These are the elements that allow the description of the password rules. It is also possible to specify a custom set of characters by surrounding it with square brackets (e.g., *[abcd]* denotes the lowercase letters from *a* to *d*). For example, to require a password with at least eight characters consisting of a mix of uppercase and lowercase letters, and at least one number, the following rule can be used:

```
required: upper; required: lower; required: digit; minlength: 8;
```

A more formal description of the grammar is shown in Figure 5.

Properties description. The ***required*** property is used when the restrictions must be followed by all generated passwords. The ***allowed*** property is used to specify a subset of allowed characters, i.e., it is used when a password is permitted to have a given character class, but it is not mandatory. If ***allowed*** is not included in the rule, all the ***required*** characters are permitted. If both properties are specified, the subspace of all ***required*** and ***allowed*** is permitted. If neither is specified, every ASCII character is permitted. The ***max-consecutive*** property represents the maximum length of a run of consecutive identical characters that can be present in the generated password, e.g., the sequence *aah* would be possible with *max-consecutive: 2*, but *aaah* would not. If multiple ***max-consecutive*** properties are specified, the value considered will be the minimum of them all. The ***minlength*** and ***maxlength*** properties denote the minimum and maximum number of characters, respectively, that a password can have to be accepted.

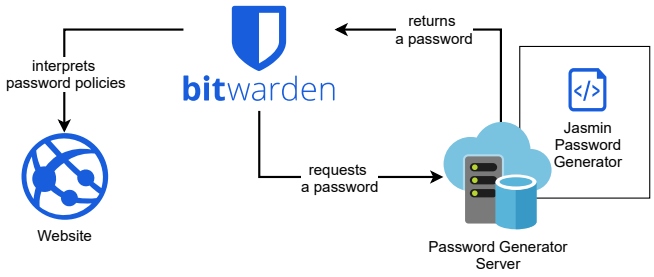

Fig. 6. Overview of Proof-of-Concept Prototype

Both numbers need to be greater than 0 and ***minlength*** has to be at most ***maxlength***; otherwise, the default length of the PM will be used.

Identifiers. Next to the *allowed* or *required* properties, we can use any of the default ***identifiers***, which describe *conventional* character classes. The identifier ***upper*** describes the character class that includes all uppercase letters, i.e., *[A-Z]*; the identifier ***lower*** describes the character class that includes all lowercase letters, i.e., *[a-z]*; the ***digits*** identifier describes the character class that includes all digits, i.e., *[0-9]*; and the ***special*** identifier describes the character class that includes *-˜!@#$%ˆ&*_+=‘|(){}[:;”’<>,.?]* and ␣. The identifiers ***ascii-printable*** and ***unicode*** describe the character classes that include all ASCII printable characters and all the unicode characters, respectively. Additionally, users of the DSL can choose to describe their custom character classes, e.g., *[aeiou]* is the character class that contains all the vowels, in lowercase.

4.2 Jasmin Password Generator

We coded our reference implementation in Jasmin [2], a framework for developing high-speed and high-assurance cryptographic software. The Jasmin programming language combines high-level and low-level 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. The Jasmin compiler is formally verified for correctness. We chose Jasmin because it is possible to automatically generate an EasyCrypt model from a Jasmin program. This ensures that when reasoning about the model, we are reasoning about the correspondent Jasmin program, making it possible to formally establish an equivalence between the Jasmin implementation and our reference implementation.

4.3 Integration With Bitwarden

An overview of our integration of the Jasmin password generator with the Bitwarden browser extension is shown in Figure 6. We first extended Bitwarden

to interpret Apple's Password Autofill Rules. We start by searching in the entire DOM for the HTML attribute *passwordrules*. When found, we parse its value. For this we used the official Apple's Javascript parser[13]. We then pass this information to the password generator component, using the browser's native messaging API. We also replaced the default password generator with our Jasmin password generator. Since in the context of the browser extension it is not possible to directly run local processes, we exposed our password generator as a RESTful service: the extension sends a POST request, with the body of the request containing the required password policy.

To demonstrate the impact of our proof-of-concept, we generated 20 test files, each one containing 1000 randomly generated passwords: 10 of these files were generated by Bitwarden's generator and the other 10 were generated by our Jasmin generator. Bitwarden's generator used its default settings — 14-character password with uppercase characters, lowercase characters, and numbers. We used the following policy, which is actually used by British government services, according to a community-updated file in Apple's repo[14]:

```
minlength: 10; required: lower; required: upper; required: digit;
required: special;
```

We checked if the passwords generated by Bitwarden satisfy this policy. All passwords failed this test, since Bitwarden's default settings do not include symbols. This is an instance of the problem discussed above, regarding users' frustration with the generation of non-compliant passwords. We then used the same approach with our Jasmin generator and found that all passwords generated satisfy the policy.

Since our extension improves the usability of Bitwarden, we submitted the code that parses the password rules and passes them to the password generator to the Bitwarden team, who has internally approved our extension and will go through a code review process to get it ready to be merged[15].

5 Related Work

To the best of our knowledge, our work is the first to address formal verification of random password generators[16]. However, the area of formal verification of security and cryptographic software has attracted much interest in recent years. Regarding implementation correctness, HACL⋆ [27] is a high-assurance cryptographic C library that has been formally verified against a readable mathematical specification in F⋆. Similarly, FiatCrypto [14] proposes a framework

[13] `https://github.com/apple/password-manager-resources/blob/main/tools/PasswordRulesParser.js`

[14] `https://github.com/apple/password-manager-resources/blob/main/quirks/password-rules.json`

[15] `https://github.com/bitwarden/browser/pull/2047#issuecomment-978846599`

[16] A search on Google Scholar shows one relevant paper [17], which is the abstract of an informal talk delivered by our team.

written in Coq for deriving correct-by-construction C code. It has been deployed in Google's BoringSSL library which is used by Chrome and Android. Targeting directly assembly, Vale [9] builds on Microsoft's Dafny and Z3 SMT prover to verify annotated assembly code. Finally, the Jasmin framework [2], that we have adopted in our development, has been previously used to produce highly-efficient certified executable code [3], combining it with security proofs in an unified framework [4].

Regarding other uses of formal verification in the domain of password security, there is previous work on creating certified password composition policy enforcement software, implemented from within the Coq proof assistant and extracted to Haskell [15]. The extracted Haskell is then compiled into a pluggable authentication module readily usable from a real Linux system. Johnson et al. [19] also used Coq to model password composition policies and verify the immunity or vulnerability of 14 password composition policies to the password guessing attacks utilised by the Mirai and Conficker botnet worms. The PassCert project[17] is exploring formal verification applied to password managers and aims to determine whether formal verification can increase users' confidence in PMs and thus increase their adoption [10, 11].

6 Conclusion

We propose a formally verified reference implementation for a Random Password Generator. We prove that, given a password composition policy, generated passwords are compliant, and we formalize the property that the generator samples the set of passwords according to a uniform distribution. In addition, we present a proof-of-concept prototype that solves the identified frustration with PMs of generating non-compliant passwords and demonstrates that our formally verified component can be integrated into a widely used PM.

As future work, we plan to fully formalize the proof of security informally discussed in Section 3.2 and to further develop the proof-of-concept prototype so that other browser-based PMs can benefit from it. We might also add support for further password composition policies (e.g. policies that require characters from at least three different classes). While generally speaking strict password composition policies are preferable, these can still generate easily guessed passwords (e.g., a policy that enforces the use of all character classes may generate the easily guessed password "P@ssw0rd") [23]. So, it might also be interesting to formalize properties regarding password strength, which would guarantee that our RPG would only generate strong passwords (according to some metric).

Acknowledgments. This work was partially funded by the PassCert project, a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019 and supported by national funds through FCT under project UIDB/50021/2020.

[17] PassCert project: `https://passcert-project.github.io`

References

[1] Nora Alkaldi and Karen Renaud. "Why do people adopt, or reject, smartphone password managers?" In: *1st European Workshop on Usable Security-EuroUSEC 2016*. 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] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. "The last mile: High-assurance and high-speed cryptographic implementations". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020.

[4] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves 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.

[5] Apple. *Customizing Password AutoFill Rules*. `https://developer.apple.com/documentation/security/password_autofill/customizing_password_autofill_rules`. [Online; accessed 31-July-2021]. 2021.

[6] Apple. *Web sites won't accept Safari generated strong passwords due to dashes or other criteria*. `https://discussions.apple.com/thread/251341081`. [Online; accessed 26-October-2021]. 2021.

[7] 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.

[8] 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.

[9] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. "Vale: Verifying high-performance cryptographic assembly code". In: *26th USENIX Security Symposium*. 2017, pp. 917–934.

[10] Carolina Carreira, João F. Ferreira, and Alexandra Mendes. "Towards Improving the Usability of Password Managers". In: *INFORUM* (2021).

[11] Carolina Carreira, João F. Ferreira, Alexandra Mendes, and Nicolas Christin. "Exploring Usable Security to Improve the Impact of Formal Verification: A Research Agenda". In: *First Workshop on Applicable Formal Methods (co-located with Formal Methods 2021)*. (2021).

[12] Sonia Chiasson, Paul C van Oorschot, and Robert Biddle. "A Usability Study and Critique of Two Password Managers." In: *USENIX Security Symposium*. Vol. 15. 2006, pp. 1–16.

[13] EA. *Password Does Not Meet Requirements*. https://web.archive.org/web/20210817105229/https://answers.ea.com/t5/EA-General-Questions/quot-Password-Does-Not-Meet-Requirements-quot/td-p/5744758. [Online; accessed 26-October-2021; archived 26-October-2021]. 2021.

[14] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. "Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises". In: *2019 IEEE Symposium on Security and Privacy, SP 2019*. IEEE, 2019, pp. 1202–1219.

[15] 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.

[16] 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.

[17] Miguel Grilo, João F. Ferreira, and José Bacelar Almeida. "Towards Formal Verification of Password Generation Algorithms used in Password Managers". In: *arXiv preprint arXiv:2106.03626* (2021).

[18] Moritz Horsch, Mario Schlipf, Johannes Braun, and Johannes Buchmann. "Password requirements markup language". In: *Australasian Conference on Information Security and Privacy*. Springer. 2016, pp. 426–439.

[19] 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.

[20] Sean Oesch and Scott Ruoti. "That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers." In: *USENIX Security Symposium*. 2020.

[21] 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.

[22] 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.

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

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

[25] Frank Stajano, Max Spencer, Graeme Jenkinson, and Quentin Stafford-Fraser. "Password-manager friendly (PMF): Semantic annotations to improve the effectiveness of password managers". In: *International Conference on Passwords*. Springer. 2014, pp. 61–73.

[26] TechNet. *Can't create local user "Password does not meet password policy requirements" - but it does.* `https://web.archive.org/web/20211026082725/https://social.technet.microsoft.com/Forums/en-US/12b06881-ea1a-403d-aafb-99bbe7d4d1b0/cant-create-local-user-quotpassword-does-not-meet-password-policy-requirementsquot-but-it?forum=win10itprosecurity`. [Online; accessed 26 October 2021; archived 26 October 2021]. 2021.

[27] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1789–1806. ISBN: 9781450349468.

[28] 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.