Certified Password Quality — A Case Study Using Coq and Linux Pluggable Authentication Modules

Certified Password Quality

A Case Study Using Coq and Linux Pluggable Authentication Modules

João F. Ferreira[1,2], Saul A. Johnson[1], Alexandra Mendes[1], and
Phillip J. Brooke[1]

[1] Teesside University, School of Computing, Middlesbrough, TS1 3BX, UK
[2] HASLab/INESC TEC, Universidade do Minho, 4704-553 Braga, Portugal
joao@joaoff.com {Saul.Johnson,A.Mendes}@tees.ac.uk pjb@scm.tees.ac.uk

Abstract. We propose the use of modern proof assistants to specify, implement, and verify password quality checkers. We use the proof assistant Coq, focusing on Linux PAM, a widely-used implementation of pluggable authentication modules for Linux. We show how password quality policies can be expressed in Coq and how to use Coq's code extraction features to automatically encode these policies as PAM modules that can readily be used by any Linux system.
We implemented the default password quality policy shared by two widely-used PAM modules: *pam_cracklib* and *pam_pwquality*. We then compared our implementation with the original modules by running them against a random sample of 100,000 leaked passwords obtained from a publicly available database. In doing this, we demonstrated a potentially serious bug in the original modules. The bug was reported to the maintainers of Linux PAM and is now fixed.

Keywords: password quality, password policy, verification, security, authentication, Coq, proof assistant, theorem prover, Linux, PAM

1 Introduction

Password quality is essential to keeping any password-protected system secure. If a password is easy to guess and an attacker gains authenticated access as a result, any security measures deployed to restrict access by unauthenticated users become irrelevant. From the perspective of the system, the attacker is indistinguishable from the legitimate user.

Without an enforced password quality policy, passwords created by users tend to be weak [11]. A password quality policy may mandate, for example, that all user passwords contain a mixture of upper case, lower case, and numeric characters in order to maximise the search space that a brute-force algorithm would need to examine in order to correctly guess a user's password. It is critical that the software that enforces these policies (the *password quality checker*) is both correct and configurable to keep up with the large body of ongoing research into password policy best-practises [7, 28, 32].

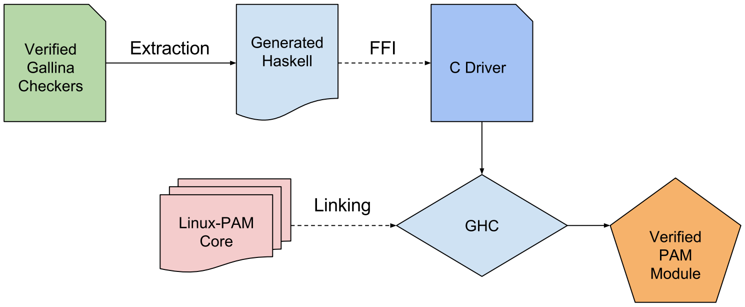

Fig. 1: An overview of the process of creating a verified PAM module.

The importance of password quality checkers makes them an ideal candidate for formal verification. Using recent advances in code generation from theorem provers, it is now possible to transform high-level verified functional implementations into certified code that can be used in place of unverified procedural code to perform password quality checking. We therefore propose the use of modern proof assistants to formally verify password quality checkers. To demonstrate this, we use the Coq proof assistant [3] to specify, implement, and verify password quality checkers. We focus on Linux PAM [25, 26], a widely-used implementation of pluggable authentication modules (PAM) for Linux. We show how we can define password quality policies in Coq and automatically encode them as Linux PAM modules that can readily be used. We document the process of extracting verified password quality assessment functions from a verified Gallina code base (Coq's specification language) into Haskell [19] and calling them via the Haskell foreign function interface (FFI) [13] from a driver written in C. Fig. 1 provides an overview of this process. We implemented several PAM modules that perform password quality checking using verified code. In particular, we implemented a module identical to the default behaviour shared by two widely-used PAM modules designed to act as password quality checkers on Linux systems: *pam_cracklib* and *pam_pwquality*. In doing this, we demonstrated a potentially serious bug in the original PAM modules. The bug was reported to Linux PAM's maintainers and is now fixed.

In Section 2, we discuss password quality checking software, focusing on Linux PAM. Section 3 is about the use of Coq to specify, implement, and verify password quality checkers. We evaluate our work in Section 4 by comparing our implementation with *pam_cracklib* and *pam_pwquality*. We also demonstrate that the flexibility of our approach allows users to create verified password policies quickly and easily. After presenting related work in Section 5, we conclude the paper in Section 6.

2 Password Quality Checking Software

Password quality checking refers to techniques used to ensure that users do not create passwords that are vulnerable to brute-force attacks or guessing by a

Passwords must:

- *Not be identical to the previous password, if any.*
- *Not be palindromic.*
- *Not be a rotated version of the old password, if any.*
- *Not contain case changes only in relation to the previous password, if any.*
- *Have a Levenshtein distance of 5 or greater from the previous password, if any (difok=5).*
- *Be at least 9 characters long (minlen=9), however:*
- *Passwords may be 1 character shorter if they contain at least 1 lower case letter (lcredit=1).*
- *Passwords may be 1 character shorter if they contain at least 1 upper case letter (ucredit=1).*
- *Passwords may be 1 character shorter if they contain at least 1 digit character (dcredit=1).*
- *Passwords may be 1 character shorter if they contain at least 1 other character (ocredit=1).*
- *This shortening of minimum length will stack, making for a minimum length of 9 - 4 = 5 for passwords containing all 4 classes.*
- *Effective minimum length is, then $M = m - c$ where M is the effective minimum length, m is the configured minimum length and c is the number of character classes present in the string.*

Fig. 2: Default policy implemented by *pam_cracklib* and *pam_pwquality*.

party with access to basic information about the user. For example, a user may be unable to create a password that is identical to their user name or email address, or that is too short. A range of other issues relate to passwords such as memorability, storage of passwords on systems, and other means to obtain passwords (such as snooping). We do not address these further in this work.

Password quality checking software often requires that an administrator provide a *password quality policy* which specifies the minimum characteristics of an acceptable password. A significant body of research is emerging that challenges conventional wisdom about what constitutes a secure password [28, 32].

Linux PAM We focus on Linux PAM [25, 26], a widely-deployed open-source application that pulls together multiple authentication-related modules into one high-level API, allowing application developers to create programs that rely on various authentication services independently of the underlying implementations. Two well-known PAM modules that are used to indicate password quality are *pam_cracklib* and *pam_pwquality*. Both modules are written in C, use the same backend, and define the same default password quality policy (see Fig. 2). Fig. 3 shows the type of code used in these modules to check whether a password is palindromic. Fig. 3a shows a pure function named `palindrome` that returns 1 if the password given is a palindrome and 0 otherwise; Fig. 3b shows how the top-level function `password_check` uses `palindrome` to check if the *new password* is a palindrome (`msg` and `_` are used for error control and internationalisation purposes respectively).

Since these modules are enabled by default in many popular Linux distributions, they are widely deployed. For example, in Red Hat Enterprise Linux 7 and in CentOS 7, the *pam_pwquality* PAM module replaced *pam_cracklib*, which was used up to version 6 as a default module for password quality checking [17]. It is estimated that CentOS is one of the most popular Linux distributions for web servers and is installed on millions of these worldwide[3].

[3] See, for example, `https://w3techs.com/technologies/details/os-linux/all/all`, and `http://www.computerworld.com/article/2468596/network-software/the-most-popular-linux-for-web-servers-is----.html`

```
static int palindrome(const char *new)
{
  int i, j;
  i = strlen(new);

  for (j = 0;j < i;j++)
    if (new[i - j - 1] != new[j])
      return 0;

  return 1;
}
```

(a) The `palindrome` function checks whether the argument string is palindromic.

```
static const char *password_check(
  pam_handle_t *pamh,
  struct cracklib_options *opt,
  const char *old, const char *new,
  const char *user)
{
  [...]
  newmono = str_lower(strdup(new));
  [...]

  if (!msg && palindrome(newmono))
    msg = _("is a palindrome");
  [...]
}
```

(b) The `password_check` function calls the function `palindrome` to check whether the proposed new password is palindromic.

Fig. 3: Two functions from *pam_cracklib.c*, one pure with only the new password accepted as a parameter, and one which drives the password checking process.

3 Verified Password Quality in Coq

We now describe how we use Coq to specify, implement, and verify password checkers. We implement checkers as pure functional programs and demonstrate Coq's flexibility by showing different ways to specify them: often, we consider the functional programs to be *functional (executable) specifications*, but we can also specify checkers *by theorem* or *by property* (i.e. axiomatically). We conclude this section by describing how verified functional implementations can be extracted as Haskell code and linked with PAM modules that can be readily used.

3.1 Types and Password Checkers

In our model, we consider passwords to be Coq strings:

```
Definition Password := string.
```

Password checkers can be seen as functions from strings to booleans (e.g. the function `palindrome` in Fig. 3a is such a function). However, we want password checkers to take into consideration more elements, such as the previous password or the user's name (see the signature of `password_check` in Fig. 3b).

In our model, we consider the user's previous password and we encode this information in the type `PasswordTransition`:

```
Inductive PasswordTransition : Set :=
  PwdTransition : (option Password) -> Password -> PasswordTransition.
```

An element of the type `PasswordTransition` represents an *old* password being changed into a *new* password. The old password is optional: if a user changes their password, the previous password is available as it must be entered to proceed; if an administrator changes the password of a user, that information is unlikely to be available. With these types defined, a password checker can be described as a function that takes a PasswordTransition and either succeeds or returns some error message. We define the type of a password checker as:

```
Definition CheckerResult := option ErrorMsg.
```

For example, a password checker that prevents passwords from being palindromes can be defined as:

```
Definition not_palindrome (pt : PasswordTransition) : CheckerResult :=
  if palindrome (new_pwd pt) then
    BADPWD: "The new password is a palindrome."
  else
    GOODPWD.
```

This defines a new password checker named not_palindrome whose behaviour is quite simple: if the new password (new_pwd pt) is a palindrome, then it should be rejected (with a specific error message). This checker depends on the function palindrome, which is discussed in the next subsection.

The reserved keywords BADPWD and GOODPWD are defined as symbolic abbreviations denoting the appropriate elements of type CheckerResult:

```
Notation GOODPWD := None.
Notation "BADPWD: msg" := (Some msg).
```

The palindrome checker uses only the new password and not the old password. This is not the general case: e.g., the old password is required when we do not want the new password to be a prefix of the old password (or vice-versa):

```
Definition prefix_old_pwd (pt : PasswordTransition) : CheckerResult :=
  NEEDS old_pwd FROM pt
    if (prefix (old_pwd pt) (new_pwd pt)) ||
       (prefix (new_pwd pt) (old_pwd pt))
    then
      BADPWD: "The new password is a prefix of the
               old password (or vice-versa)"
    else
      GOODPWD.
```

This password checker returns an error if the old password (old_pwd pt) is a prefix of the new password (new_pwd pt) or vice-versa. The checker depends on the function prefix, which is discussed in the next subsection. The body of this checker is prefixed by a new construct expressing that the old password is required to define the checker: NEEDS old_pwd FROM pt. The definition of NEEDS means that if the old password is undefined (e.g. if the administrator is changing the password of a normal user), then the check is disabled. Further, the function old_pwd is being exposed to the checker as a *local function*. This provides a safer way to access the old password, because using old_pwd pt without prefixing it with the NEEDS construct will result in a type error (caught at compilation time). In other words, the function old_pwd is only available in contexts where the old password is defined, thus avoiding conditional boilerplate code that checks whether the old password is defined.

3.2 Specification, Implementation, and Proofs

An advantage of defining password checkers in a proof engineering environment such as Coq is that we can prove properties about implementations. For example, if we want to prove that `prefix_old_pwd` is skipped when the old password is undefined, we can state and prove a lemma as follows:

```
Lemma prefix_old_pwd_undefined: forall (pt: PasswordTransition),
    old_pwd_is_undefined(pt) = true  ->  prefix_old_pwd(pt) = GOODPWD.
Proof.
  intros. unfold old_pwd_is_undefined in H.
  (* Case analysis *)
  destruct pt. destruct o.
   (* Case 1 (trivial): old password is defined *)
   - congruence.
   (* Case 2: old password is undefined *)
   - unfold prefix_old_pwd. simpl. auto.
Qed.
```

The lemma simply states that if the old password is undefined[4], then the checker `prefix_old_pwd` is disabled (i.e. it accepts all passwords). The proof is by case analysis and is made simple by using tactics such as `congruence`, `simpl`, and `auto`.

In the context of our work, the most important aspect to verify is functional correctness. We have seen above that password checkers are functions from `PasswordTransition` to `CheckerResult` that normally depend on inner pure functions. For example, the checker `not_palindrome` depends on `palindrome` and `prefix_old_pwd` depends on `prefix`. In general, when defining password checkers, we are interested in proving that the inner pure functions are correct. In the remainder of this section, we discuss different approaches to specify password checkers. The point of showing different specification approaches is to demonstrate that writers of verified password checkers can use their preferred style of specification (e.g. functional programmers will probably prefer to write functional executable specifications).

Functional (executable) specifications As we are using a high-level functional programming language to encode password checkers, we can give direct implementations of constructive or executable specifications [30, 31]. E.g., the following definition of palindrome acts both as specification and implementation:

```
Definition palindrome (s : string) : bool :=
  s ==_s (string_reverse s).
```

This definition is an implementation (i.e. it can be executed), but it also describes the notion of palindrome: an arbitrary string `s` is a palindrome if and only if `s` is the same as its reverse. Most programmers would be satisfied with this

[4] `old_pwd_is_undefined(pt)` is defined to return true when the old password is undefined and false otherwise

specification, but because we are in a proof engineering environment, we can prove further properties; an example is the following lemma stating that the function that reverses a string is involutive.

```
Lemma string_reverse_involutive : forall (s : string),
  string_reverse (string_reverse s) = s.
Proof.
  induction s as [| c s’].
  (* Base case *)
   - simpl. reflexivity.
  (* Inductive step *)
   - simpl. rewrite (string_reverse_unit (string_reverse s’) c).
     rewrite IHs’. auto.
Qed.
```

The proof is by induction and uses the lemma `string_reverse_unit`, which states that for all strings s and characters c, we have:

$$\mathit{string\_reverse}(\mathit{string\_append}(s, c)) = \mathit{string\_append}(c, \mathit{string\_reverse}(s))$$

Specification by theorem A proof assistant like Coq also allows us to specify functions by capturing their specifications as theorems. E.g., the function `prefix`, used in the password checker `prefix_old_pwd`, can be specified as:

```
Theorem prefix_correct : forall s1 s2 : string,
 prefix s1 s2 = true  <->  substring 0 (length s1) s2 = s1.
```

This theorem states that a string `s1` is a prefix of a string `s2` if and only if `s1` is the substring of length `length s1` starting at position 0 of `s2` (i.e., for $k =$ `length s1`, the string composed by the k leftmost characters of `s2` is `s1`). This is proved in Coq's standard library.

Specification by property Strong specifications usually demand a greater proving effort: proofs are normally more complex and it is often the case that deeper knowledge of the proof assistant is required.

In some cases, it may be easier or desirable to prove properties that do not fully specify the implementation, but nevertheless increase our confidence in its correctness. For example, suppose that we define the Hamming distance [14, 15] between two strings of equal length as follows:

```
Fixpoint hamming_distance (a b : string) : option nat :=
  match a, b with
    | EmptyString, EmptyString => Some 0
    | String ca a’, String cb b’ =>
      match hamming_distance a’ b’ with
        | None => None
        | Some n => Some ((nat_of_bool (negb (ca ==_a cb))) + n)
      end
    | _, _ => None
  end.
```

Instead of fully specifying this function, we increase our confidence in this implementation by proving properties the Hamming distance satisfies. For example:

```
Lemma hamming_distance_undefined_for_different_lengths : forall (a b : string),
  length a <> length b  <->  hamming_distance a b = None.

Lemma hamming_distance_defined_for_same_length : forall (a b : string),
  length a = length b  ->  hamming_distance a b <> None.

Lemma hamming_distance_zero_for_identical : forall (s: string),
  hamming_distance s s = Some 0.
```

3.3 Password Policies and Code Extraction

Our framework mimics the behaviour of the PAM modules *pam_cracklib* and *pam_pwquality* in that password quality policies are lists of password checkers executed successively. E.g., the policy shown in Fig. 2 is defined as follows:

```
Definition pwd_quality_policy :=
  [ diff_from_old_pwd ; not_palindrome ; not_rotated ;
    not_case_changes_only ; levenshtein_distance_gt 5 ;
    credits_length_check 8 ].
```

This list, together with all its contents, is extracted into Haskell code by using Coq's code extraction mechanism [24]. Finally, the extracted Haskell code is linked with a C driver to create a PAM module that calls the Haskell code via Haskell's foreign function interface (FFI) [13]. In short, the C code calls each password checker with a password transition and reports the result to the user.

4 Evaluation

In this section, we evaluate our work by comparing the newly implemented verified PAM module to the original in terms of behaviour, performance, and compiled executable size. We describe the bug discovered in the original module, and demonstrate that the flexibility of our approach allows users to create verified password policies quickly and easily.

4.1 Experimental Setup

Using Vagrant, a virtual machine running Ubuntu 16.04 "Xenial" 64-bit with Coq v8.6 and the Glasgow Haskell Compiler v7.10.3 installed was created to provide a consistent testing environment [23]. An unmodified instance of this machine was used for every test run.

A random sample of 100,000 passwords was obtained from a publicly available database of ten million leaked passwords [5] using a Python script. An instance of the test machine was then configured to use each module in turn as the password quality checker for its native *passwd* executable, which handles user password changes. A set of shell scripts was created to run each password through this executable one at a time and record the results, which consist of feedback from the active PAM module about the strength of the submitted password. As the script terminates *passwd* after the first password entry, no actual password change was performed as the password must be entered twice (for confirmation) in order to effect one. Importantly, the passwords were checked on their own merit and not in the context of a password change; that is, the old password in use before the attempted password change was not taken into account during password quality checking. As a result of this, any password quality checks that compare the new password to the old password in any way were not in effect. This raw data was passed through a Python script which consolidated it into a CSV file ready for further analysis using spreadsheet software.

The behaviour of the verified module was then compared to the original module. All dictionary checks were disabled in the original module (and omitted from the verified module) prior to testing. All source code was maintained under source control on GitHub [22].

4.2 Experiment 1: Comparison with PAM Modules *pam_cracklib* and *pam_pwdquality*

The verified PAM module was first configured and built to implement the default policy shared by both *pam_cracklib* and its successor *pam_pwquality* (shown in Fig. 2 and encoded as shown in Section 3.3).

As expected, the verified module behaved identically to the original, accepting 56574 of the passwords in the database (that is, deeming them secure enough) with absolute consistency between them (i.e. the same passwords were accepted or rejected).

Aside from the behaviour of the module itself and whether or not it is written using verified code, there are other factors that may be considered when deciding on the most suitable module to use on any one system. For example, performance and executable size. In order to compare the performance of the verified module to the original module, each run of *passwd* during the experiment was timed and averaged to calculate an average checking time per password (Table 1).

Module	Description	Avg. Time
pam_cracklib_nodict	Original C implementation of pam_cracklib with dictionary check disabled.	0.00926278s
pam_basic_pwd_policy	Verified module built with the default pam_cracklib default policy enabled (without dictionary check).	0.011845369s

Table 1: Average execution time for each test run.

The average checking time for the verified module is around 1.28 times that of the unverified C module in all cases, but this difference is not as drastic as had been anticipated, considering that many algorithms in use within the verified module are not nearly as efficient as those in the original (compare the inefficient — yet easier to reason about — definition of palindrome shown in Section 3.2 to the implementation shown in Fig. 3a).

With regard to executable size, it is unsurprising that the compiled verified module is significantly larger than the original module (Table 2). The verified module is linked against several dependencies from both the Haskell and C standard libraries. The authors recognise, however, that on non-critical storage-constrained systems, it may be inconvenient to use an executable around 9 times the size of its unverified counterpart when its behaviour is expected to be identical.

File Name	Description	File Size
pam_cracklib_nodict.so	Original C implementation of pam_cracklib with dictionary check disabled.	22384 bytes
pam_basic_pwd_policy.so	Verified module built with the default pam_cracklib default policy enabled (without dictionary check).	189688 bytes

Table 2: File size comparison between the original and verified modules.

4.3 Experiment 2: Increasing Password Entropy

Research into password complexity [16] has shown that it may become almost mandatory for users to create longer passwords that contain a good mixture of uppercase and lowercase letters, numbers, and symbols. It would not be unreasonable, then, for a system administrator to enforce a policy mandating that no passwords have more than two characters of the same class (i.e. type) in a row in an effort to boost entropy (see Table 3 for examples).

Password	Accepted	Reason
1234Password	No	More than one number in a row, more than one lowercase letter in a row.
1Ll4m4!Gg	Yes	No more than one number, uppercase letter, lowercase letter or symbol in a row.
correcthorsebatterystaple	No	More than one lowercase letter in a row.
Ab4kUs#!	No	More than one symbol in a row.

Table 3: Example of the status of different hypothetical passwords under the proposed policy.

In order to accomplish this using *pam_cracklib*, the *maxclassrepeat* option must be set to 1. After configuring the original *pam_cracklib* and the verified module in this way (using the policy from Fig. 2 with the additional constraint that no two consecutive characters may be of the same class), the test was run again over the same password database. In this case, the modules did not perform identically.

While the verified module predictably accepted only a tiny minority (371) of passwords, the original module exhibited exactly the same behaviour as before and accepted 56574 passwords. This result demonstrated the effects of a bug in *pam_cracklib*, specifically a check done inside *pam_cracklib.c* on line 411:

```
if ( opt->max_class_repeat > 1  && sameclass > opt->max_class_repeat) {
  return 1;
}
```

Rather than checking if the option `max_class_repeat` is set to a number greater than zero, the check is done against 1 instead (see highlighted code). This has the consequence of disabling the check entirely, which contradicts the documentation for the option and any intuition on the part of the system administrator.

This issue was raised on the Linux PAM GitHub repository [18], along with a pull request containing the fix. A project maintainer reviewed it to their satisfaction and merged the fix into the official repository, to be distributed in future releases. After the fix had been applied, the *pam_cracklib* module was compiled and tested again against the password database, this time functioning consistently with the verified module.

4.4 Experiment 3: A Simple Policy

To demonstrate the flexibility of our approach, we show that it is possible to quickly and easily compile a password quality checker PAM module drawing on specific research findings. Kelly et al. [21] suggest that the use of the *basic16* password policy (16 alphabetic characters) creates passwords that are more resilient against brute-force attacks than policies such as *complex8* which allows for shorter (length 8), but more complex passwords containing a mixture of cases, numbers, and symbols.

The verified module was quickly reconfigured, rebuilt, and reinstalled with this new, very simple policy in place. In code, we simply alter the list of password quality checkers to apply only a length check and nothing more, before extracting the Coq code to Haskell and rebuilding the C driver:

```
Definition pwd_quality_policy := [
  plain_length_check 16
].
```

The policy makes use of the `plain_length_check` function that evaluates a password on length alone:

```
Definition plain_length_check (len : nat) (pt : PasswordTransition)
  : CheckerResult := if length (new_pwd pt) >=? len then GOODPWD
                     else BADPWD: "The new password is too short.".
```

The accompanying `plc_correct` lemma and proof certify that this function behaves correctly:

```
Lemma plc_correct: forall (len : nat) (pt : PasswordTransition),
      plain_length_check len pt = GOODPWD
<-> is_true (length (new_pwd pt) >=? len).
Proof. repeat (split; unfold plain_length_check;
               destruct (length (new_pwd pt) >=? len); crush). Qed.
```

In this case, because the function is very simple, the implementation is as complex as its specification. However, in general, this is not the case (see, for instance, the examples in Section 3). The proof is based on the definition of the function and a case analysis on the length of the new password. It also depends on the *crush* tactic from [10]. On running this newly-configured checker over the password database, 970 passwords were accepted while the rest were shorter than 16 characters in length and therefore rejected. Interestingly, the original *pam_cracklib* and *pam_pwquality* libraries can not be configured in this way without making changes at the source code level and recompiling, as various checks (`palindrome` being one example) cannot be disabled through configuration alone. While our approach also requires recompilation of the verified module, the scope of the required source code changes (modification of one list) is so small that it arguably amounts to little more than a configuration change. In this way, our approach is demonstrably more flexible than that taken by the original modules.

5 Related Work

To the best of our knowledge, this is the first effort in creating verified password quality checkers. The closest related work on provably improving the reliability of authentication systems is the body of work on verification of authentication protocols. For example, the work presented in [27] and [12] uses CSP and PVS to analyse and verify authentication properties. A very popular automatic cryptographic protocol verifier is ProVerif [4]. Uses of ProVerif include the verification of a user authentication protocol named oPass [29] and security properties of mutual-authentication and key-exchange protocols [6].

The work presented in this paper has been motivated by recent advances that make practical the verification of system security components [1]. In particular, we were inspired by approaches that are based on extracting (or generating) code directly from proof assistants. An example is FSCQ [8, 9], the first file system with a machine-checkable proof (using Coq). Similar to what we do, a Haskell implementation is extracted using Coq's extraction feature. Two additional examples are the implementation of a conference management system [20] and of a distributed social media platform [2], where code generation was also used to extract correct Scala implementations from Isabelle specifications.

6 Conclusion

Through this work, we have used the proof assistant Coq to create verified password quality checkers in the form of PAM modules with at least as much functionality (aside from dictionary checks) as *pam_cracklib* and *pam_pwquality* which are already widely deployed. We identified a potentially serious bug and we demonstrated that our framework can be used to easily create new certified password quality policies.

Despite the successes, limitations remain. While we use a code extraction approach that substantially reduces the size of the unverified code base, it does not eliminate it entirely. Some low-level unverified C code must still be written in order to call the extracted code in a useful context. Importantly, while the Gallina code is verified, the authors are not aware of any correctness proof of Coq's code extraction mechanism. Executable size is also greatly increased in the verified modules almost by an order of magnitude, which may place serious limitations on its use by storage-constrained systems.

The collection of proofs for the verified checkers is being constantly improved as part of an ongoing verification effort as we investigate potential future work in this area (see [22]). In particular, we aim at making most proofs as simple and automatic as possible. Nevertheless, as we demonstrated, the framework allows the creation of new policies that are completely verified. This work focuses on the specific and important area of verified password checking and we believe that it lays a foundation for further research in this area.

Future Work A domain-specific language (DSL) is in development as a direct successor to this research which will allow Linux system administrators to quickly and easily express their ideal password quality policy and produce a verified password quality checker PAM module in one compilation step. We anticipate that this will offer a great deal of flexibility beyond the simple configuration options offered by existing password quality checking PAM modules.

In continuing this work, we hope to substantially reduce the size of the unverified C driver by stripping out functionality that is not absolutely necessary or that has been made redundant by our verification efforts. We also plan to verify other aspects of the PAM modules such as configuration option parsing as well as extend the functionality of the verified password quality checking code to include dictionary checks. An examination of the feasibility of adding Unicode support is also planned.

References

[1] Andrew W Appel. “Modular Verification for Computer Security”. In: *IEEE 29th Computer Security Foundations Symposium (CSF)*. 2016, pp. 1–8.

[2] Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. “CoSMeDis: A Distributed Social Media Platform with Formally Verified Confidentiality Guarantees”. In: *Security and Privacy (SP)*. 2017.

[3] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development – Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media, 2013.

[4] Bruno Blanchet et al. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules." In: *CSFW.* Vol. 1. 2001, pp. 82–96.

[5] Mark Burnett. *Today I Am Releasing Ten Million Passwords.* `https://xato.net/today-i-am-releasing-ten-million-passwords-b6278bbe7495`. Accessed: 2017-04-26. 2015.

[6] Ran Canetti and Jonathan Herzog. "Universally composable symbolic analysis of mutual authentication and key-exchange protocols". In: *Theory of Cryptography Conference.* Springer. 2006, pp. 380–403.

[7] National Cyber Security Centre. *Password Guidance: Simplifying Your Approach.* `https://www.ncsc.gov.uk/guidance/password-guidance-simplifying-your-approach`. Accessed: 2017-04-26. 2016.

[8] Tej Chajed, Haogang Chen, Adam Chlipala, M Frans Kaashoek, Nickolai Zeldovich, and Daniel Ziegler. "Certifying a file system using crash Hoare logic: correctness in the presence of crashes". In: *Communications of the ACM* 60.4 (2017), pp. 75–84.

[9] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. "Using Crash Hoare logic for certifying the FSCQ file system". In: *Proceedings of the 25th Symposium on Operating Systems Principles.* ACM. 2015, pp. 18–37.

[10] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013.

[11] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. "Password strength: An empirical analysis". In: *INFOCOM.* IEEE. 2010, pp. 1–9.

[12] Bruno Dutertre and Steve Schneider. "Using a PVS embedding of CSP to verify authentication protocols". In: *Theorem Proving in Higher Order Logics* (1997), pp. 121–136.

[13] Sigbjorn Finne, Inc Fergus Henderson, Marcin Kowalczyk, Daan Leijen, Simon Marlow, Erik Meijer, Simon Peyton Jones, and Malcolm Wallace. *The Haskell 98 Foreign Function Interface 1.0 An Addendum to the Haskell 98 Report.* 2002.

[14] Richard W Hamming. *Coding and Theory.* Prentice-Hall, 1980.

[15] Richard W Hamming. "Error detecting and error correcting codes". In: *Bell Labs Technical Journal* 29.2 (1950), pp. 147–160.

[16] Philip G Inglesant and M Angela Sasse. "The true cost of unusable password policies: password use in the wild". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM. 2010.

[17] Mirek Jahoda, Robert Krátký, Martin Prpič, Tomáš Čapek, Stephen Wadeley, Yoana Ruseva, and Miroslav Svoboda. *Red Hat Enterprise Linux 7 Security Guide.* `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Security_Guide/index.html`. Accessed: 2017-04-26. 2017.

[18] Saul Johnson. *Behavior of maxclassrepeat=1 inconsistent with docs.* `https://github.com/linux-pam/linux-pam/issues/16`. Accessed: 2017-03-31. 2017.

[19] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.

[20] Sudeep Kanav, Peter Lammich, and Andrei Popescu. "A conference management system with verified document confidentiality". In: *International Conference on Computer Aided Verification.* Springer. 2014, pp. 167–183.

[21] Patrick Gage Kelley, Saranga Komanduri, Michelle L Mazurek, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Julio Lopez. "Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms". In: *Security and Privacy (SP).* IEEE. 2012, pp. 523–537.

[22] Software Reliability Lab. *Verified PAM Cracklib.* `https://github.com/sr-lab/verified-pam-cracklib`. Accessed: 2017-04-05. 2017.

[23] Software Reliability Lab. *Verified PAM Environment.* `https://github.com/sr-lab/verified-pam-environment`. Accessed: 2017-03-30. 2017.

[24] Pierre Letouzey. "Extraction in Coq: An overview". In: *Conference on Computability in Europe.* Springer. 2008, pp. 359–369.

[25] Andrew G Morgan and Thorsten Kukuk. *The Linux-PAM Module Writers' Guide.* 2010.

[26] Vipin Samar. "Unified login with pluggable authentication modules (PAM)". In: *Proceedings of the 3rd ACM conference on Computer and Communications Security.* 1996, pp. 1–10.

[27] Steve Schneider. "Verifying authentication protocols in CSP". In: *IEEE Transactions on Software Engineering* 24.9 (1998), pp. 741–758.

[28] Richard Shay, Saranga Komanduri, Adam L Durity, Phillip Seyoung 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), p. 13.

[29] Hung-Min Sun, Yao-Hsin Chen, and Yue-Hsun Lin. "oPass: A user authentication protocol resistant to password stealing and password reuse attacks". In: *IEEE Transactions on Information Forensics and Security* 7.2 (2012), pp. 651–663.

[30] Simon Thompson. "Functional programming: executable specifications and program transformations". In: *ACM SIGSOFT Software Engineering Notes.* Vol. 14. 3. 1989, pp. 287–290.

[31] Joost Visser, José Nuno Fonseca Oliveira, LS Barbosa, João Fernando Ferreira, and Alexandra Mendes. "CAMILA revival: VDM meets Haskell". In: *1st Overture Workshop.* University of Newcastle TR series. 2005.

[32] Leah Zhang-Kennedy, Sonia Chiasson, and Paul van Oorschot. "Revisiting password rules: facilitating human management of passwords". In: *APWG Symposium on Electronic Crime Research (eCrime).* IEEE. 2016, pp. 1–10.