Passlab: A Password Security Tool for the Blue Team

Saul A. Johnson[1]

[1] School of Engineering, Computing and Digital Technologies, Teesside University

What is a Red Team?

- The CNSS Glossary says it best.
- When it comes to exploiting weak user passwords, the red team (the team performing the penetration test) has a whole host of tools to work with.
- Password cracking tools like Hashcat/John the Ripper, scripts to run online guessing attacks/credential stuffing attacks etc.

"A group of people authorized and organized to emulate a potential adversary's attack or exploitation capabilities against an enterprise's security posture."

—Committee on National Security Systems (CNSS) Glossary[1]

What is a Blue Team?

- Password security tools are a lot thinner on the ground when it comes to defending against password guessing attacks.
- My Ph.D. aims to contribute formal methods to synthesise and evaluate password composition policies—sets of rules around user password creation designed to encourage stronger passwords.

“A group of individuals that conduct operational network vulnerability evaluations and provide mitigation techniques to customers who have a need for an independent technical review of their network security posture.”

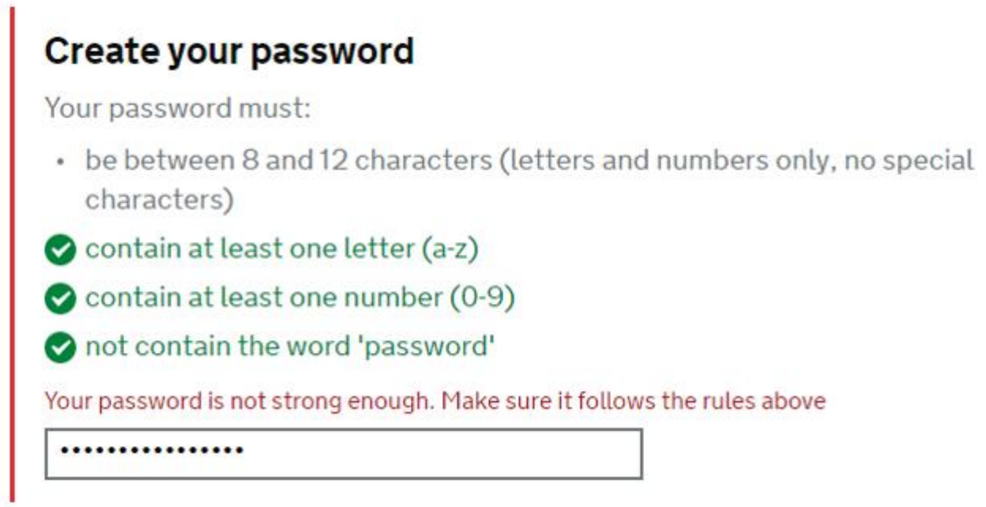
—Committee on National Security Systems (CNSS) Glossary[1]

Some Background!

What are password composition policies, and what are their desirable security properties?

Password Policies: You Know them Already!

- Password composition policies have become a fact of life when creating/changing passwords.
- They are designed with the intention of making password guessing attacks less likely to succeed by encouraging users to choose passwords that are harder to guess.
- The vast majority of these policies out there today are unfit for purpose.
- The password composition policy behind the form on the right belongs/belonged to the British Revenue and Customs agency HMRC. Is there anything wrong with it?



Create your password

Your password must:

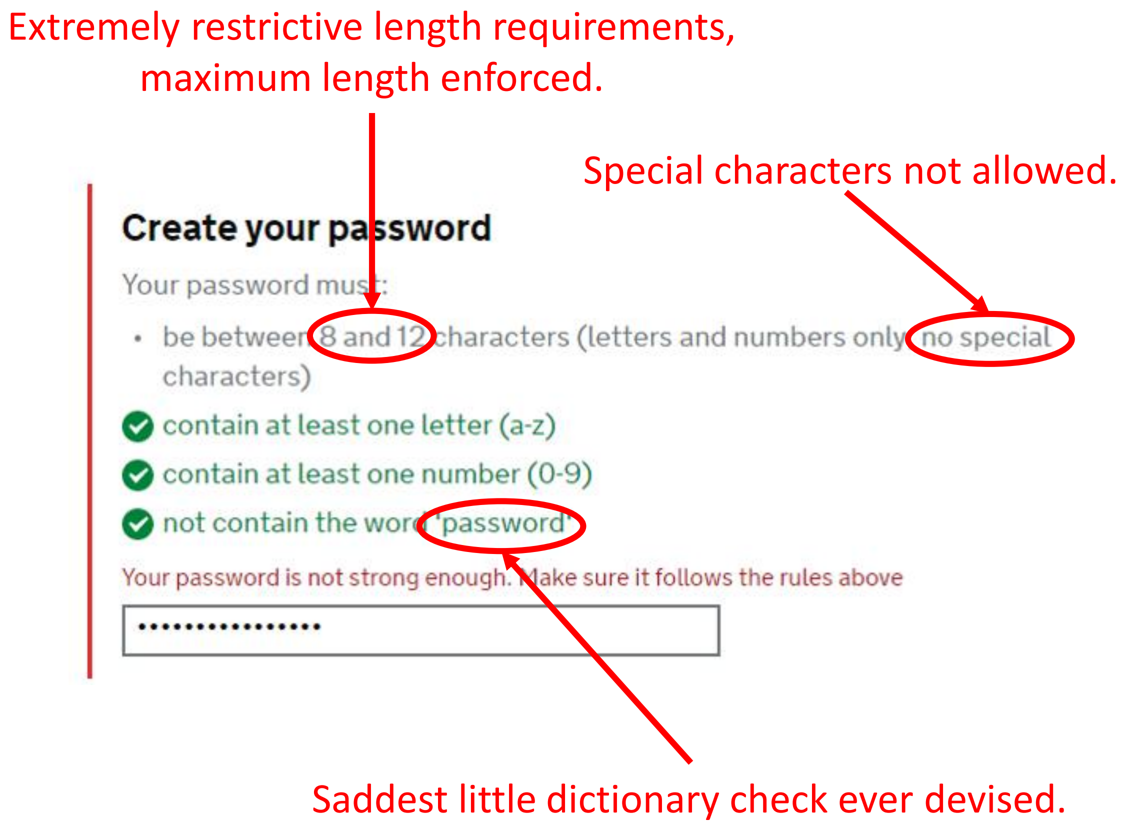
- be between 8 and 12 characters (letters and numbers only, no special characters)
- ✓ contain at least one letter (a-z)
- ✓ contain at least one number (0-9)
- ✓ not contain the word 'password'

Your password is not strong enough. Make sure it follows the rules above

················

Answer: Yes, Yes There Certainly Is

- The password space is extremely restricted by the length constraints and limitations on character set.
- One of the most common mistakes users make when creating their passwords is to overuse dictionary words. This "dictionary" check prohibits one word.
- There is never an excuse for prohibiting passwords containing certain characters or passwords that are "too long". All passwords should be hashed to a fixed-length string anyway, so why should the website care?



Create your password

Your password must:

- be between 8 and 12 characters (letters and numbers only, no special characters)
- ✓ contain at least one letter (a-z)
- ✓ contain at least one number (0-9)
- ✓ not contain the word 'password'

Your password is not strong enough. Make sure it follows the rules above

Extremely restrictive length requirements, maximum length enforced.
Special characters not allowed.
Saddest little dictionary check ever devised.

Choice and Enforcement: Related But Distinct

- To effectively employ a password composition policy on a system, we must first choose a policy in an informed way, and be able to justify that choice.
- Then, we must ensure that this policy is enforced correctly on the system.
- Our aim is the development of a tool that puts both of these things within the reach of a system administrator with little to no background in password security or formal methods.

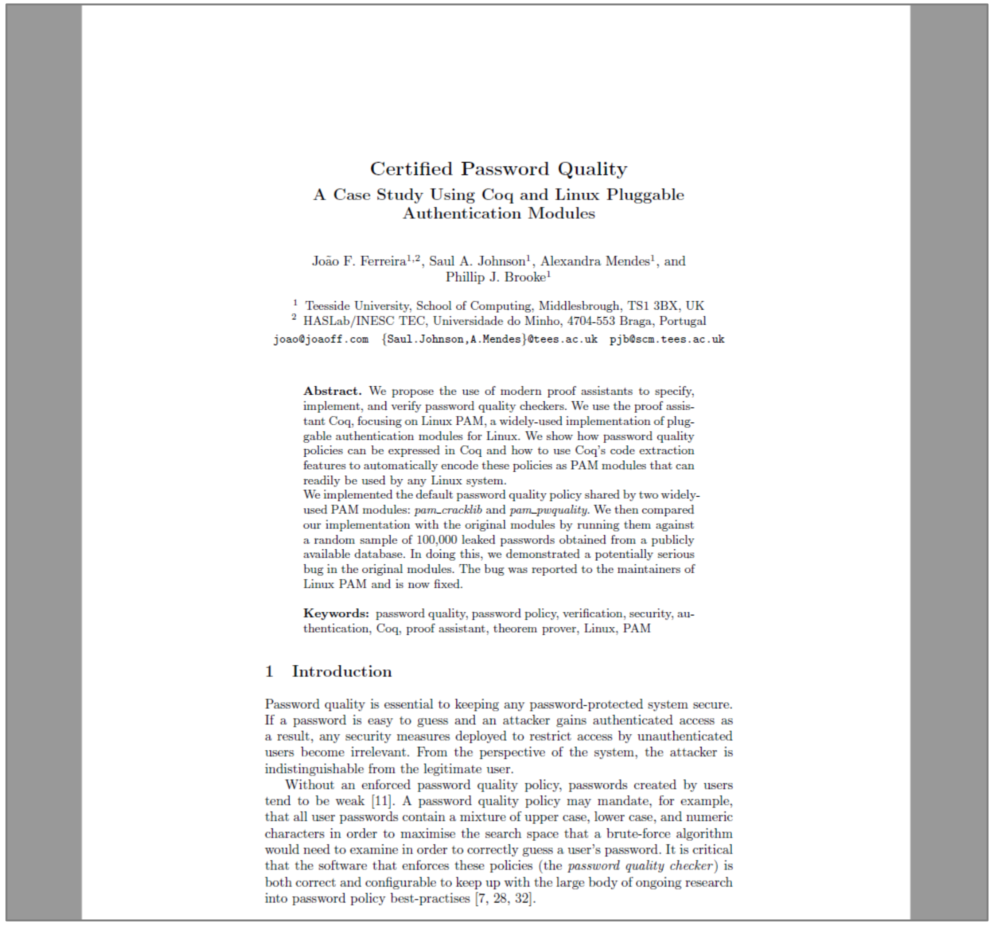
Verified Password Composition Policy Enforcement

Developing formally verified password composition policy enforcement software.

Certified Password Quality

- Our first paper, Certified Password Quality[2] was presented at iFM 2017 in Turin, Italy. It implements verified password composition policy enforcement by demonstrating the encoding of password composition policies using the Coq proof assistant.
- These policies could then be extracted to Haskell and used from a C driver program, allowing them to be used as pluggable authentication modules on real Linux systems!
- In doing this, we uncovered a bug in the original modules dating all the way back to (we think) 1996/7! We submitted a PR containing a fix, which was merged:

  https://github.com/linux-pam/linux-pam/issues/16



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].

Serenity: Extending Our First Work

- Our domain-specific language (DSL) Serenity has been in the works for a while now and builds on our first piece of work, to permit system administrates to build password composition policy enforcement software that is correct by construction.
- The language is embedded within the Coq proof assistant, and its building blocks are formally verified.
- It's also intuitive enough that system administrators can straightforwardly express their intended policy with minimal training. DSLs have a recognised advantage in being easy to adopt[3].

An Example

- As an example, consider a password composition policy that prohibits password that are palindromes (i.e. read the same forwards as backwards).
- The Serenity DSL contains a checker for this, which is itself formally verified. The palindrome function is shown on the left, and one of the correctness properties of its constituent functions (string_reverse) is shown above.
- In providing a DSL for specifying password composition policies with formally verified building blocks, we permit system administrators to create correct-by-construction software enforcing them.

```
(* String reversal is involutive. *)
Theorem string_reverse_involutive
    : ∀ (s : string),
string_reverse (string_reverse s) = s.
Proof.
    (* Proof omitted here for brevity. *)
Qed.

(* A string is a palindrome if it equals
 * itself, reversed. *)
Definition palindrome (s : string) : 𝔹 :=
match string_dec s (string_reverse s) with
    | left _ => true
    | right _ => false
end.
```

Serenity: An Example Policy

```
Definition comprehensive8 :=
  (enforce new_pwd (min length 8)
    "New password must be at least 8 characters long!")
  /*\ (enforce new_pwd (min count_upper 1)
    "New password must contain an uppercase letter!")
  /*\ (enforce new_pwd (min count_lower 1)
    "New password must contain a lowercase letter!")
  /*\ (enforce new_pwd (min count_digit 1)
    "New password must contain a digit!")
  /*\ (enforce new_pwd (min count_other 1)
    "New password must contain a symbol!")
  /*\ (prohibit new_pwd palindrome
    "New password cannot be a palindrome!").
```

Justifiable Password Composition Policy Selection

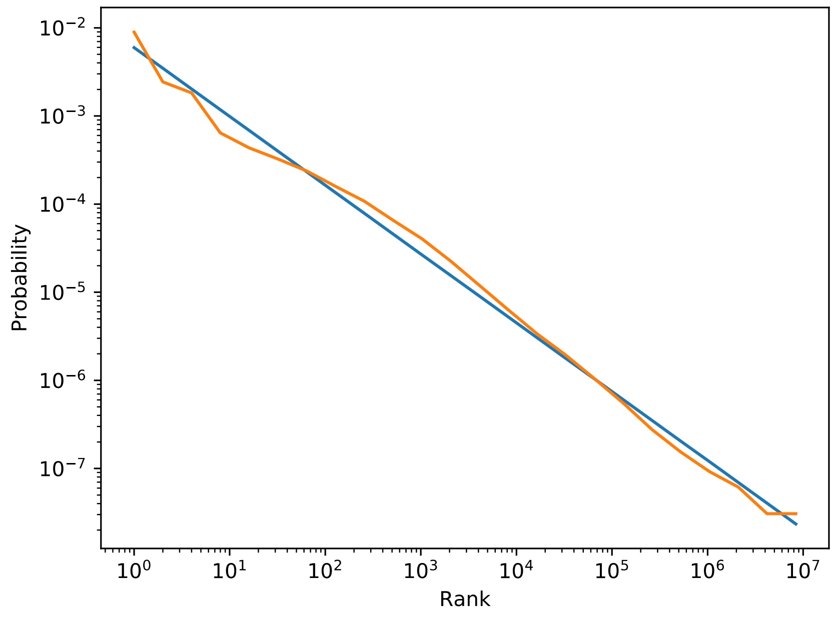
Building a system for automatic, justified and privacy-preserving password composition policy choice.

Measuring Password Policy Impact

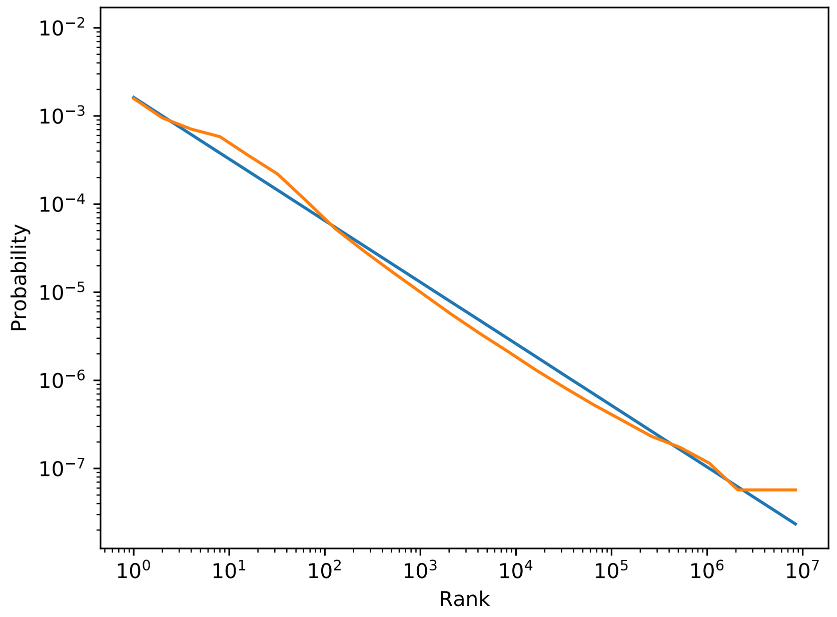
- Formal methods for evaluating the impact of password policies on human-chosen passwords must be informed by human-chosen passwords!
- Password selection varies (though not as much as you'd think) across systems, demographics, age groups[4], so we need to select a large, breached password database that is as representative as possible of our user base.
- Password diversity is the key here. We want users to select passwords from the available password space as uniformly as possible to protect against guessing attacks[5, 6, 7].

More Varied Passwords, More Secure System!

RockYou (Weaker Policy): Steeper Curve/Less Uniform Distribution (Length 5)

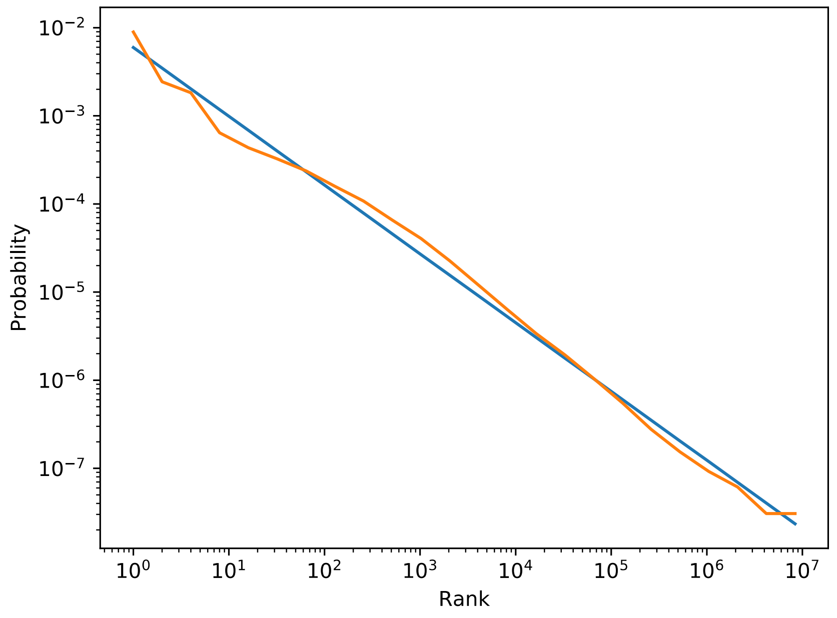


000webhost (Stronger Policy): Shallower Curve/More Uniform Distribution (Length 6, 1 Digit)



We Don’t Need to See User Passwords!

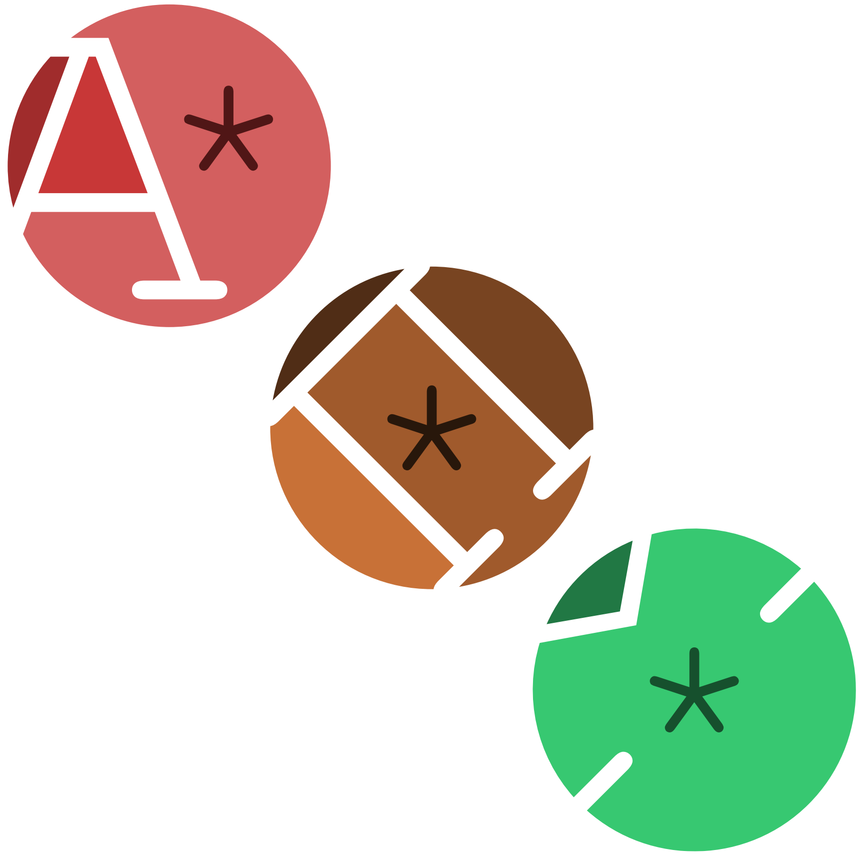
- A key finding so far in our work is that we don’t actually need to have access to user passwords in order to justify our password policy choice. This is great for avoiding the propagation/sharing of password data and preserving user privacy.
- Password distributions tend to follow Zipf’s law[5, 6]. A few passwords are chosen very often, and many passwords are chosen rarely, with an exponential fall-off.
- This means all we need to rank password policies is the equations that fit the distributions they induce. See the blue line on the right.



Skeptic: Automatic, Justified and Privacy-Preserving Password Policy Choice

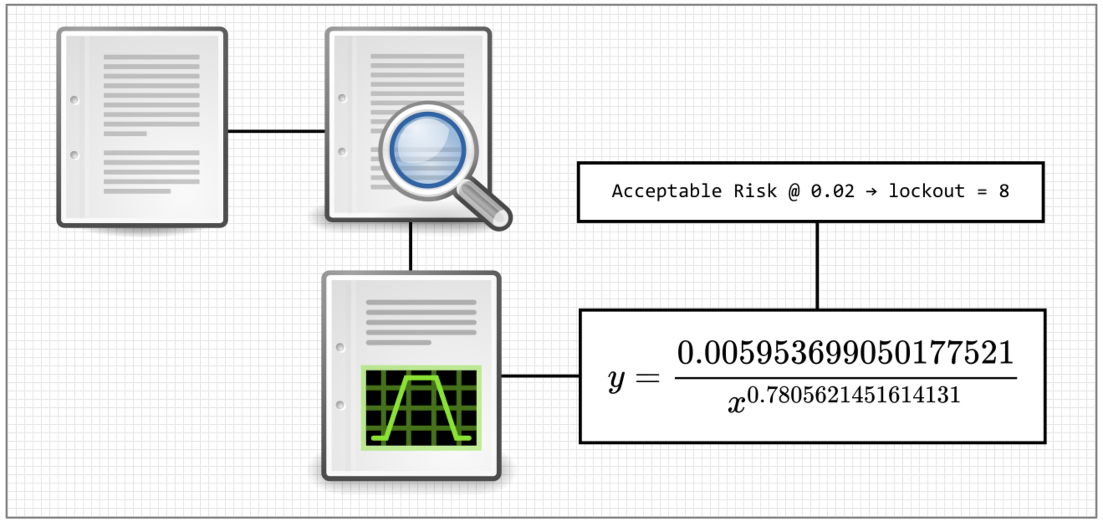
Our project culminates in *Skeptic*, a 3-part system for automatically choosing a password composition policy.

- **Authority:** A verified core written in Coq that filters a password data dump according to a policy.
- **Pyrrho:** A user behaviour model that simulates users choosing different passwords in response and fits power-law equations to the resulting distributions. Written in Python.
- **PaCPAL:** Password composition policy assertion language. A DSL that allows system administrators to easily extract results from this data automatically. Written in Idris.



Pulling it All Together: A User-Friendly Tool

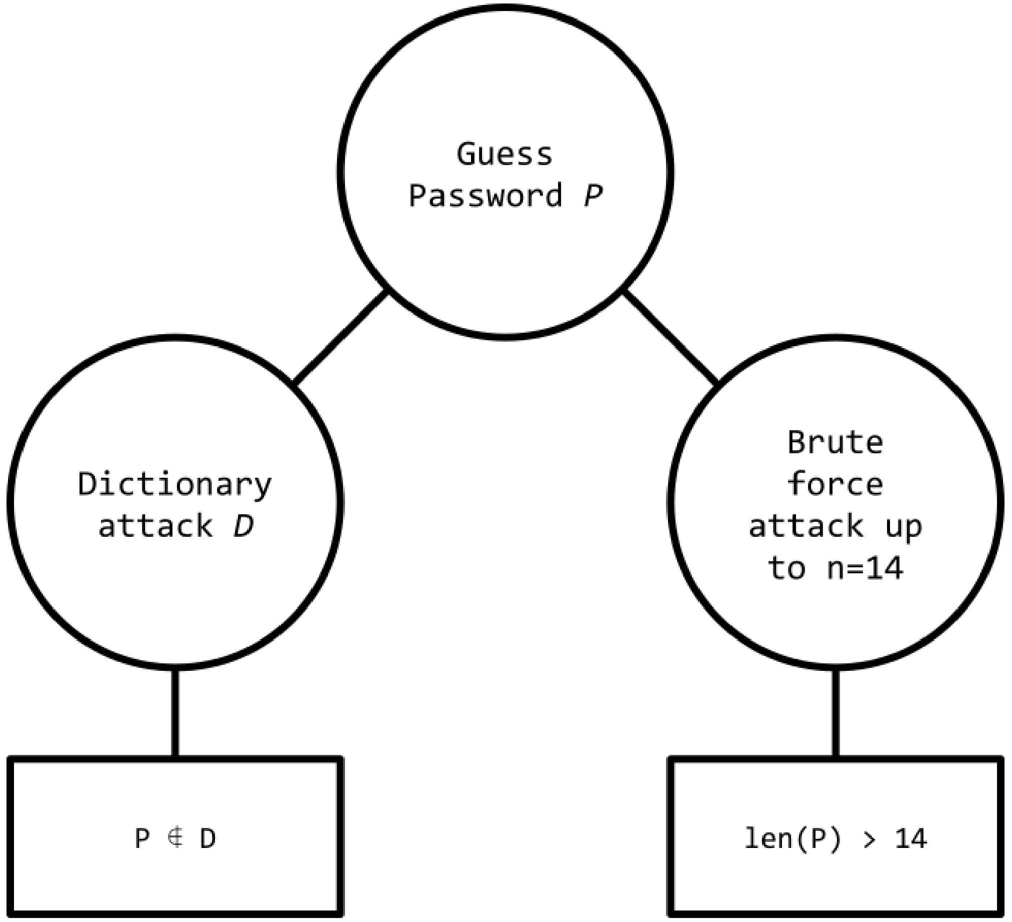
- *Passlab* is a proposed piece of software, written in Java, for pulling together the Skeptic framework into an intuitive UI.
- Development is already well underway!
- It uses a visual editor to allow users to evaluate password policies by examining their effect on real-world password datasets.
- On the right, data flows from a raw data source (e.g. a downloaded password data dump) through a formatting node and to a Zipf model node, which fits a power-law equation to the data.
- From here, we can synthesise a "lockout policy"[9]. How many incorrect password entry attempts should we allow before locking the account down to keep probability of breach below 0.02?
- Password policy filtration nodes are next to be implemented!



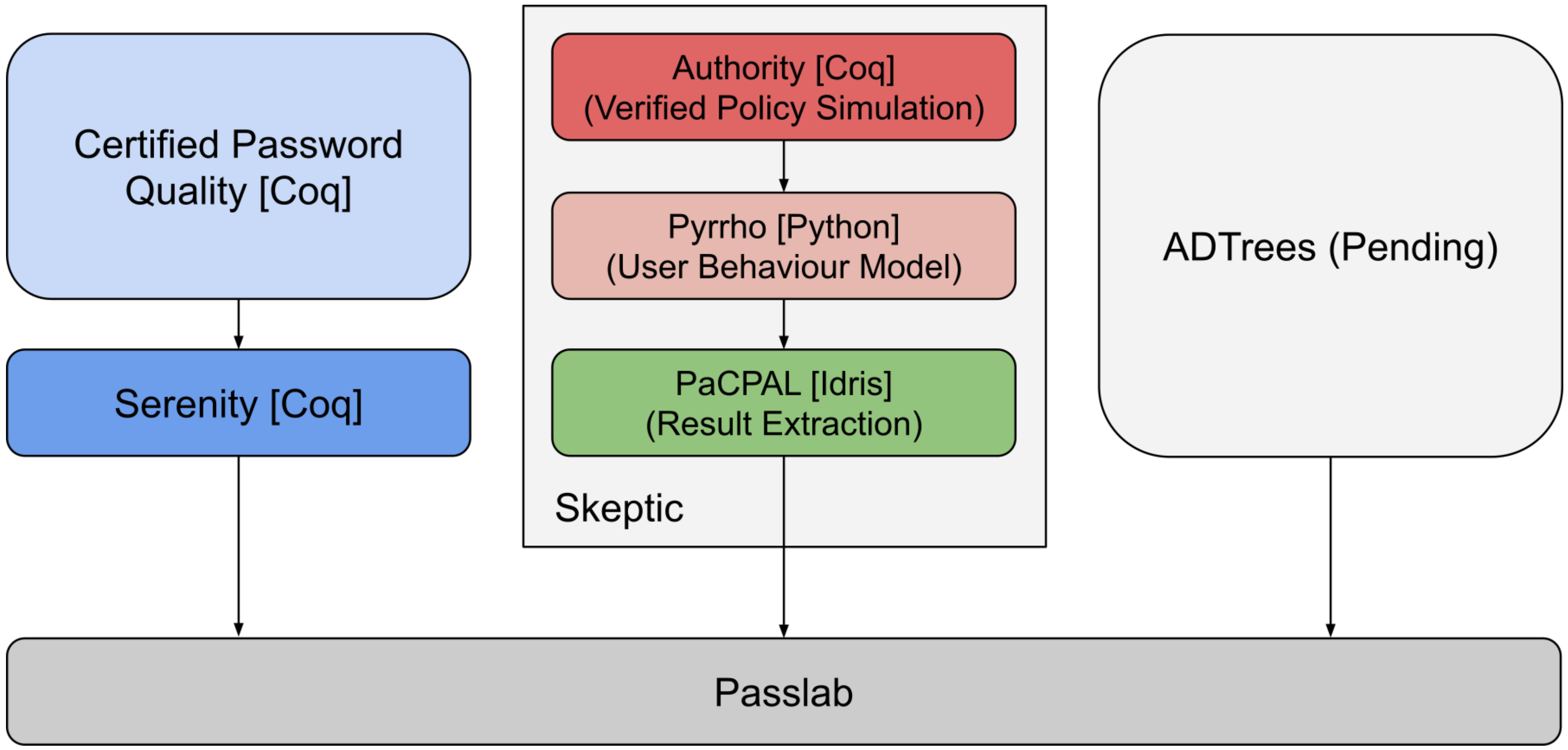
$$y = \frac{0.005953699050177521}{x^{0.7805621451614131}}$$

Pulling it All Together: A User-Friendly Tool (cont.)

- We plan to add support for synthesis of password composition policies using Attack-Defence Trees[10].
- This will allow users to tailor password composition policies to specific guessing attacks (or classes of guessing attack) that they expect their systems to encounter.
- On the right here is an example of what such an ADTree might look like. The attacker has a goal to guess password P, using a dictionary attack D or a brute-force attack up to length-14 passwords.
- To mitigate this attack, we use defence nodes mandating that password P is not in D, and that the length of the password is greater than 14.



Making Sense of All These Tools!


Certified Password Quality [Coq]
Serenity [Coq]
Authority [Coq] (Verified Policy Simulation)
Pyrrho [Python] (User Behaviour Model)
PaCPAL [Idris] (Result Extraction)
Skeptic
ADTrees (Pending)
Passlab

References

1. Committee on National Security Systems (CNSS) Glossary, Committee on National Security Systems Instruction (CNSSI) No. 4009
2. Ferreira J.F., Johnson S.A., Mendes A., Brooke P.J. (2017) Certified Password Quality. In: Polikarpova N., Schneider S. (eds) *Integrated Formal Methods. IFM 2017.* Lecture Notes in Computer Science, vol 10510. Springer, Cham
3. A. Bariic, V. Amaral and M. Goulão, "Usability Evaluation of Domain-Specific Languages," *2012 Eighth International Conference on the Quality of Information and Communications Technology*, Lisbon, 2012, pp. 342-347.
4. J. Bonneau, "The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords," *2012 IEEE Symposium on Security and Privacy*, San Francisco, CA, 2012, pp. 538-552.
5. David Malone and Kevin Maher. 2012. Investigating the distribution of password choices. In *Proceedings of the 21st international conference on World Wide Web (WWW '12)*. ACM, New York, NY, USA, 301-310.
6. D. Wang, H. Cheng, P. Wang, X. Huang and G. Jian, "Zipf's Law in Passwords," in *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 11, pp. 2776-2791, Nov. 2017.
7. Sean M. Segreti, William Melicher, Saranga Komanduri, Darya Melicher, Richard Shay, Blase Ur, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Michelle L. Mazurek. 2017. Diversify to survive: making passwords stronger with adaptive policies. In *Proceedings of the Thirteenth USENIX Conference on Usable Privacy and Security (SOUPS '17)*, Mary Ellen Zurko, Sonia Chiasson, and Matthew Smith (Eds.). USENIX Association, Berkeley, CA, USA, 1-12.
8. Jeremiah Blocki, Saranga Komanduri, Ariel Procaccia, and Or Sheffet. 2013. Optimizing password composition policies. In *Proceedings of the fourteenth ACM conference on Electronic commerce (EC '13)*. ACM, New York, NY, USA, 105-122. DOI: https://doi.org/10.1145/2492002.2482552
9. Dinei Florêncio, Cormac Herley, and Paul C. Van Oorschot. 2014. An administrator's guide to internet password research. In *Proceedings of the 28th USENIX conference on Large Installation System Administration (LISA'14).* USENIX Association, Berkeley, CA, USA, 35-52.
10. Kordy B., Mauw S., Radomirović S., Schweitzer P. (2011) Foundations of Attack–Defense Trees. In: Degano P., Etalle S., Guttman J. (eds) *Formal Aspects of Security and Trust. FAST 2010.* Lecture Notes in Computer Science, vol 6561. Springer, Berlin, Heidelberg